A University of Sussex DPhil thesis

Available online via Sussex Research Online:

http://sro.sussex.ac.uk/

This thesis is protected by copyright which belongs to the author.

This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the Author

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the Author

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given

Please visit Sussex Research Online for more information and further details

Computing multi-scale organizations built through assembly

Gregory M. Studer

Submitted for the degree of D.Phil.

University of Sussex

August 6, 2011

Declaration

I hereby declare that this thesis has not been submitted, either in the same or different form, to this or any other university for a degree.

Signature:

Acknowledgements

I am indebted first of all to my advisor Inman Harvey, who was responsible for shepherding my early interests in multi-scale phenomena as they repeatedly twisted and turned. I count myself lucky to also have a whole host of supportive colleagues - Lucas Wilkins, Jose Fernandez-Leon, Nick Tomko, Matthew Egbert, Renan Moioli, Bruno Santos, and the rest of the CCNR - your help has made this thesis as complete as it could be and infinitely more accessible. A special mention should also be made of Cristiano Solarino: our coffee discussions were a major factor in clarifying the ideas within and pushing me in new and interesting directions.

Of course, supporting this research at a more fundamental level is my family and my wife Sara. Each one of you has always been incredibly supportive of my often incomprehensible goals. You guys are the best.

Computing multi-scale organizations built through assembly

Gregory M. Studer

Summary

The ability to generate and control assembling structures built over many orders of magnitude is an unsolved challenge of engineering and science. Many of the presumed transformational benefits of nanotechnology and robotics are based directly on this capability. There are still significant theoretical difficulties associated with building such systems, though technology is rapidly ensuring that the tools needed are becoming available in chemical, electronic, and robotic domains. In this thesis a simulated, general-purpose computational prototype is developed which is capable of unlimited assembly and controlled by external input, as well as an additional prototype which, in structures, can emulate any other computing device. These devices are entirely finite-state and distributed in operation. Because of these properties and the unique ability to form unlimited size structures of unlimited computational power, the prototypes represent a novel and useful blueprint on which to base scalable assembly in other domains.

A new assembling model of Computational Organization and Regulation over Assembly Levels (CORAL) is also introduced, providing the necessary framework for this investigation. The strict constraints of the CORAL model allow only an assembling unit of a single type, distributed control, and ensure that units cannot be reprogrammed - all reprogramming is done *via assembly*. Multiple units are instead structured into aggregate computational devices using a procedural or developmental approach. Well-defined comparison of computational power between levels of organization is ensured by the structure of the model. By eliminating ambiguity, the CORAL model provides a pragmatic answer to open questions regarding a framework for hierarchical organization.

Finally, a comparison between the designed prototypes and units evolved using evolutionary algorithms is presented as a platform for further research into novel scalable assembly. Evolved units are capable of recursive pairing ability under the control of a signal, a primitive form of unlimited assembly, and do so via symmetry-breaking operations at each step. Heuristic evidence for a required minimal threshold of complexity is provided by the results, and challenges and limitations of the approach are identified for future evolutionary studies.

Submitted for the degree of D.Phil.

University of Sussex

August 6, 2011

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Systems assembled from huge numbers of tiny components are the basis of all living and non-living processes in the natural world. Through basic interactions *structure* is generated, and composed from these structures more structure; from the smallest particles we can observe to the largest galaxy clusters. What is truly fascinating about the natural world is the incredible diversity and dynamism at any scale one looks, all of it fundamentally generated from the same sorts of basic physical interactions, everywhere. For example, our own body is an amalgamation of nanotech devices, organized through multiple levels into a macroscopic body, controlled via an impossibly intricate microscopic electrical system. Only recently have scientists begun to emulate in artificial ways this quintessential trick of nature - the generation and control of dynamic structure over a broad range of scales.

The notion of system hierarchy via assembled components is not new, particularly in the life sciences. Organisms are amazing constructions of systems of systems built from a more-or-less single kind of part, recognized in the development of cell theory in the 1800s. Cells themselves are built from chemical and atomic organizations, which fundamentally rely on a small number of basic chemicals. Though these ideas have been well-known for the latter half of the 20th century (Maynard Smith & Szathmáry, 2004), the development of artificial chemistry and artificial life in the late 1980s made the *constructive* and *computational* understanding of life and related processes a topic of active research. Because living systems are composed of multi-scale interactions, a major challenge of this project is to "create a formal framework for synthesizing dynamical hierarchies at all scales" (Bedau et al., 2000; Lenaerts et al., 2005; Bedau, 2007).

The control and generation of dynamic, multi-scale devices is not of interest only to biology, as recent convergent advances in nanotechnology, chemistry, electronics, and robotics attest (Whitesides, 2002). At the macroscale, assembling robotic devices are theorized to have extreme flexibility in the types of tasks they can accomplish, while potentially being more robust and cheaper (Yim et al., 2007a, 2009). As machines get smaller, toward the micro- and nano- scale, building machines upwards from simpler, stochastic assembling parts is one of the only practical ways to generate complex devices, be they mechanical (White & Yim, 2007; Tolley et al., 2008; Yim et al., 2009) or chemical (Gómez-López & Stoddart, 2002; Drexler et al., 2007; Gazit, 2007). In all of these research areas, the main promise and challenge lies in building and harnessing structures of

many simple parts organized over multiple scales. This remains difficult, despite demonstrated technical capability to build assembling parts in each domain. Robotic devices of thousands of units remain elusive (Yim et al., 2009), the construction of complex, multi-part nanotech devices is still on the horizon (Whitesides et al., 1991; Whitesides, 2002; Drexler et al., 2007), and the gap between remains comparatively unknown territory. General principles of controllable assembly across scales are required, divorced from particular environments, along with constructive "blueprints" for assembling parts generally applicable in many areas.

1.1 Contributions

Toward these goals, this thesis introduces a framework, based on a distributed *computational* notion of interaction, which supports well-defined and constructive Computational Organization and Regulation over Assembly Levels - the CORAL model. By formalizing the central question of "how can we generate and control dynamic structure across scales?" to a system restricted by shared constraints of assembly and well-defined composite structures, it is possible to unambiguously describe devices organized at higher structural levels. This model then provides a framework for the design of structurally and computationally scalable prototype units, while ensuring a minimum of assumptions about the initial unit scale or environment. These prototypes demonstrate for the first time that unlimited, dynamic constructive power over unlimited scales is achievable (and easily controllable) in a computational sense, requiring a base unit with fewer states than a single CPU register. *The CORAL model and these prototypes provide the framework and blueprints for uniquely scalable and controllable assembling systems.*

1.1.1 Dynamic, scalable assembly and computation

At the heart of these capabilities are the ideas of distributed, concurrent, and composable computation. The primary tool used by the CORAL model is Petri nets and the subset of capacity-1 Petri nets called condition-event (C/E) nets (Reisig, 1992; Petri, 1996). Petri nets are a well-known model of distributed computation developed in the 1960s by Carl Adam Petri (Petri, 1962), and can be viewed as a token-based game or, equivalently, synchronized collections of finite state machines. Mathematically, Petri nets can be represented as a bipartite graph of "places" and "transitions," where places may be marked with tokens and token flow is synchronized between multiple places by transitions. Synchronization is a primitive Petri net operation, which is used as the basis of the assembly operation in the CORAL model. The operation of units when assembled is changed only by synchronizing particular transitions with others, allowing the construction of composite units which *are themselves Petri nets*. A more detailed description and explanation is provided in Chapter 3.

The major innovation allowed by this architecture is this tight integration of internal unit computation and external unit interaction while maintaining basic notions of topology. In every assembling system, external interactions between units eventually become the internal processing inside assembled structures. In previous models of assembling systems with topological structure, such as amorphous computing models (Abelson et al., 2000), artificial chemistries (Mayer & Rasmussen, 1998; Banzhaf et al., 1999; Dorin, 2000; Ono & Ikegami, 2001; Ewaschuk & Turney, 2006; Hutton, 2007), or robotically-inspired systems (Studer & Lipson, 2006; Sayama, 2009), the

atomic parts are represented in a qualitatively different way than the composite structures. Often, for example, passive individual "molecules" float freely in a 2D or 3D world. Over time these form semi-stable aggregates, which are then interpreted as dynamic structures at different levels using various heuristics. These structures, however, must be described in qualitatively different ways than the molecules themselves, and may additionally include fluid boundaries, internal state, and mobility. Comparisons between the atomic parts and hybrid environmental structures consequently become difficult, if not impossible, clouding ideas of complexity and hierarchical organization. This is also the case in natural systems, where larger objects typically interact differently from the components of which they are made.

Progress can be made, however, by meeting at the middle: atomic units and composite assemblies can both be modeled using internal state and finite (though potentially large) boundaries. It is also possible to restrict multi-unit structures to interact only directly through the mechanisms of their constituent units, particularly when building new assembling machines. Artificial assembling devices are designed naturally this way, and it is hypothesized that other, scale-dependent interactions of structures can be more easily understood as additions to the inherently scale-independent core system. By linking this model of hierarchical assembly, inspired by abstract models of dynamical hierarchy, to a simple control scheme and general topological space, the *construction* and *control* of huge structures becomes possible. Essentially, this thesis documents a search for the smallest realistic substrate for unlimited computational development.

1.1.2 Research summary

With the basic ideas of assembly across scales as a backdrop, the novel contributions of the work in this thesis are listed individually below:

- a framework for assembling systems which allows the direct computational comparison of assembled devices across many orders of magnitude of assembly - the CORAL model - while also including basic realistic assumptions such as conservation of mass and finite connectivity (Chapter 3),
- using this framework, the first example of a simple assembling prototype with well-defined, distributed, and dynamic controllability in structures of *unlimited* size and *arbitrary* logical computation - the NOR unit (Chapter 4),
- the first example of a simple assembling prototype with the ability to build dynamic computing artifacts of *unlimited* size and *arbitrary* computational complexity, including its own controller, in an environment preserving the basic notions of realism above - the C/E unit (Chapter 5),
- and an original investigation using evolutionary algorithms to design simple examples of scalable assembling (pairing) units, including a comparison of various algorithms and genotype encodings as a base for future work (Chapter 6).

1.2 Clarifications

Before beginning the thesis proper, it is helpful to outline a few issues in terminology and concepts which may inadvertently misdirect or confuse the reader. The study of assembling systems is a highly interdisciplinary field, and different vocabulary and approaches are more popular in

different areas. In addition, it is useful to clarify exactly what is meant by an assembling system, since readers may initially have different impressions as to what the term means.

1.2.1 Assembling systems

The notion of an assembling system is defined here as a *collection of objects or units which can build composite objects called structures*. This definition is similar or more general than those given elsewhere for assembly and self-assembly (Andeen, 1997; Adleman, 2000; Whitesides, 2002; Banzhaf, 2004). The focus on objects instead of organizations is deliberate, as the work presented here does not consider dynamically stable structures as entities themselves. A pragmatic approach is taken instead where every novel topological structure is considered a different type of assembled device, which allows the direct comparison of structures with units when using the CORAL assembly model. This definition also supports recursion; structures can be relabeled as units to form a second-order assembling system.

1.2.2 Assembly and self-assembly

In much related literature, the terms "self-assembling system" or "self-assembly" are used to describe systems compatible with the very general definition given above. The definition of self-assembly in (Banzhaf, 2004) is a slight exception, as it limits the scale of the process to a single "stage" or level, with "self-formation" allowing recursive constructs. Assembling systems described in this thesis exhibit self-formation in this context.

The prefix "self-" usually indicates that individual devices at some point act under limited external control, but the extent of this idea varies. For example, is a system self-assembling if smart but immobile parts are attached by stupid external robots (Werfel et al., 2006), or if the external environment (Krishnan et al., 2007) powers and directs most assembly itself? Developmental systems (Harding & Banzhaf, 2008), in particular, do not fall neatly into this category. It is acknowledged here that the distinction between self- and normal assembly is an inherently fuzzy border, made more so by complications establishing whether a structure of multiple units is a unit itself. The preferred term in this work is simply "assembly," to indicate that the interesting philosophical issues related to identity and control are not addressed. Again, a pragmatic approach is used where unrealistic individual manipulation is forbidden, but realistic, limited broadcast communication and control is not.

1.2.3 Assembly and self-organization

A similar problem of definition exists for the term "self-organization." Though there have been many attempts to precisely define self-organization and emergence (e.g. (Ashby, 1962; Haken, 1987; Baas, 1994; Banzhaf, 2004)), the terms still encompass different ideas to different people and so are avoided for clarity. Related work in multi-unit simulated systems that build structure is often presented as an example of self-organization. The claim is not challenged, but this type of research is also considered here as an example of assembly. Similarly, multi-agent systems are here considered instances of multi-unit systems if the agents form larger structure. One property many models called self-organizing do share is demonstrated organization at only a single higher

descriptive level - the "self-organized" level. This was the impetus behind research into multiple "levels of organization," strongly related to dynamical hierarchies and the work in this thesis.

1.2.4 Assembly and dynamical hierarchies

Dynamical hierarchies are a flexible concept in dynamical systems theory and artificial life, where the underlying description of some system is refactored into more granular representations without losing essential descriptive ability. Though there is not general agreement on the best way of going about this process, dynamical hierarchies have been identified in many types of systems: smooth dynamical systems (Jacobi, 2005), population-based systems (Rowe et al., 2005), discrete chemical interactions (Dittrich & di Fenizio, 2007), and finite state automata (Nehaniv & Rhodes, 2000), amongst others. Assembling systems do not necessarily constitute dynamical hierarchies, however most work constructing such hierarchies naturally integrates assembly either explicitly or implicitly, and it has been hypothesized that assembly is a required component for such hierarchies to emerge (Bedau et al., 2000). Because the CORAL model allows well-defined recursive structures of different sizes with comparable computational dynamics, dynamical hierarchies based on subsets of computational behavior are naturally observed.

A caveat - it is also true that few if any observed system hierarchies in the physical world are strictly child-to-parent or parent-to-child. Information flows bi-directionally through many scales - for example, particular molecules are required for an organism to move a large body, but the results of directed motion include changes to these molecules, potentially mediated through a highly complex nervous system and digestive processes. Circular dependence is ubiquitous, and an active area of research. The terms "hierarchy," "structural level," and "level of assembly" must be read here and throughout this thesis somewhat ambiguously when applied to real systems, implying not that all interactions conform to some sort of tree-like organization but only that some interactions do.

1.3 Thesis overview and structure

A brief summary of the chapters of the thesis is presented below as a guide to the reader.

Chapter 2 begins with a brief history of assembly research, dividing the discussion between the early pioneering work in a variety of fields and the modern interdisciplinary synthesis which exists today. Particular attention is paid to modeling approaches in the different fields, which motivate the choices made in the CORAL model. The idea of reconfiguration in response to input is also discussed, which gives the primary motivation for the global-broadcast input mechanism used throughout the other chapters.

The CORAL model is fully introduced in Chapter 3. As described above, the model is based on C/E net units, which, in combination, become larger C/E net units. An example unit is presented which generates chain and loop structures, but assembly is fundamentally limited, motivating the two assembling prototypes of the next two chapters.

In Chapter 4, the first prototype is introduced - an assembling unit implementing a logical Not-OR (NOR) operation. By attaching itself to other NOR units in particular structures, meta-units implementing the NOR operation can also be built, and themselves directed into even larger constructions. NOR units therefore are the first instance of a realistic assembling model to demon-

strate unlimited assembly over arbitrary orders of magnitude in size. These mega-structures can themselves be controlled, and placed into other structures that compute arbitrary logical expressions, the output of which can drive a variety of other actions not modeled in the core CORAL framework.

The second type of assembling unit, the C/E unit, is able to build arbitrarily large structures which can emulate any other computational device. Described in Chapter 5, these units are also capable of building their own C/E net structure at larger scale. The meta-nets constructed are potentially more powerful than, but composed of, the individual unit interactions, and are assembled via an interesting distributed graph assembly algorithm called "turtle assembly."

Chapter 6 presents results from an experiment into the evolutionary design of hierarchically assembling devices. Such an investigation is interesting in itself, as there have been few examples evolving the design of assembling devices using simulated evolution, and results comparing different types of algorithms and genotype encodings are provided. In addition, it is known from the NOR unit example that the CORAL model supports unlimited hierarchical assembly. The results from this thesis therefore suggest that the limited hierarchy demonstrated thus far in virtual ecosystem and artificial chemistry models incorporating evolution is a result of hierarchy being difficult to evolve using current evolutionary algorithms, and is not a fundamental limitation of abstract environments.

In Chapter 7, a synthesis of the ideas from the three previous chapters relating to recursive assembly is presented. Other perspectives on assembly and self-organisation are compared with the demonstrated scalable assembly results and the engineering trade-offs inherent in the approach are discussed. The chapter ends with a discussion of modeling in systems with dynamic organisational capabilities.

Finally, Chapter 8 summarizes the results put forward in the previous chapters, identifies limitations of the work, and recommends ideas for extensions.

Chapter 2

Background and Motivation

Research into the design of assembling systems currently spans a broad swathe of disciplines, with the term "assembly" used to describe systems of both complex interacting machines and organisms to comparatively simple reactive simulations. This large scope is not simply a consequence of language but also of the models, which tend to have similar roots to research begun soon after the advent of computer science in the 1930s. In this chapter these origins are briefly introduced, giving the background of relevant research in artificial chemistries and robotics which inspired and inform the assembly model developed for this thesis. Much recent work has addressed aspects of assembly at multiple organizational levels and various means of building assemblies in a controllable way, and a contribution of this thesis is to link these previously disparate ideas together using a model that can demonstrate both.

2.1 Origins of research into assembling systems

Explicit study into artificial assembling systems began in the late 1940s, when the explosion of electronics research first made such devices conceivable. At Los Alamos National Laboratory, John Von Neumann was investigating the potential self-replication of mechanical devices using ideas from automata theory, newly developed by pioneering computer scientists of the 1930s. The core of his idea, reprinted in 1966 from earlier lectures, was this:

> ...[O]ne imagines automata which can modify objects similar to themselves, or effect syntheses by picking up parts and putting them together, or take synthesized entities apart. ...Draw up a list of unambiguously defined elementary parts. Imagine there is a practically unlimited supply of these parts floating around in a large container. One can then imagine an automaton functioning in the following manner: It is also floating around in this medium; its essential activity is to pick up parts and put them together, or, if aggregates of parts are found, to take them apart. (Von Neumann, 1966)

Informed by the current understanding of self-assembling systems in biology, Von Neumann was aware that real self-assembling machines would probably be linked in complex ways to the environment in which they interact. Understanding the general processes of self-assembly required

sidestepping this problem using some sort of abstract, prototypical environment, and in discussions with a colleague Stanislaw Ulam he was led to analogous mathematical models of crystal assembly (Ulam, 1950; Beyer et al., 1985). By marrying a regular crystal grid with automata theory, Von Neumann designed the first cellular automata (CA) - a regular grid of computing elements which interact locally and synchronously. As an abstraction to answer questions about mechanical replication, each grid *cell* in a CA originally corresponded with some simple mechanical atom, highly limited in function but powerful when acting cooperatively. Using this model (and others), Von Neumann was able to pose the following question:

> Can one build an aggregate out of such elements in such a manner that if it is put into a reservoir, in which there float all these elements in large numbers, it will then begin to construct other aggregates, each of which will at the end turn out to be another automaton exactly like the original one? (Von Neumann, 1963)

Von Neumann provided his own answer in the the publication of the universal CA constructor (as well as other universal models) (Von Neumann, 1966). This original constructor was an unwieldy structure of thousands of cells (an impressive mathematical construction in an era before desktop computing) but was further refined several years later to only eight and four states in two respective theses by Edgar Codd (Codd, 1968) and Edwin Roger Banks (Banks, 1971). Universal constructive power, the ability of particular constructs to build any other structure, implies that a system supports self-replication, though simpler self-replicators exist, as was shown a decade later by Christopher Langton (Langton, 1984). Self-replication is not a particular focus of the research presented in this thesis, but it was and is a major driver of assembly research. All self-replicating devices require some sort of multi-part assembly (perhaps by definition), and investigation into biological reproduction informs assembly research to this day.

Although originally designed for investigating mechanical assembly, cellular automata have been and continue to be used as a model for discrete, distributed populations in a variety of fields too numerous to describe here. They also continue to be used as a tool to investigate distributed assembly processes, and are related to many current models of assembling or distributed systems discussed later in this chapter.

2.2 Early assembling systems

A second major strand of assembly thought began similarly in the 1950s with the pioneering cyberneticist Gordon Pask, and was reintroduced to the artificial intelligence (AI) mainstream much later in the early 1990s (Pask, 1958a; Cariani, 1993; Bird & DiPaolo, 2008). Instead of using mechanical devices, Pask devised a self-assembling "ear" from a chemical soup of iron particles in sulphuric acid. Too small and numerous to individually direct, the particles instead could be influenced to form long chains by external electrical signals, becoming a chemical system tuned by the user (or itself) to build structure (Figure 2.1). These structures then acted as analog computers, filtering multimodal information from the environment and controlling the production of new structure. This type of analog work was superseded with the rise of digital computation, and only slowly have computer scientists, inspired by biological and chemical structure, continued asking many of the same questions. To this day, Pask's ear is notable in both the simplicity of the assembling unit and the huge scale over which assemblies were formed, from molecular units to

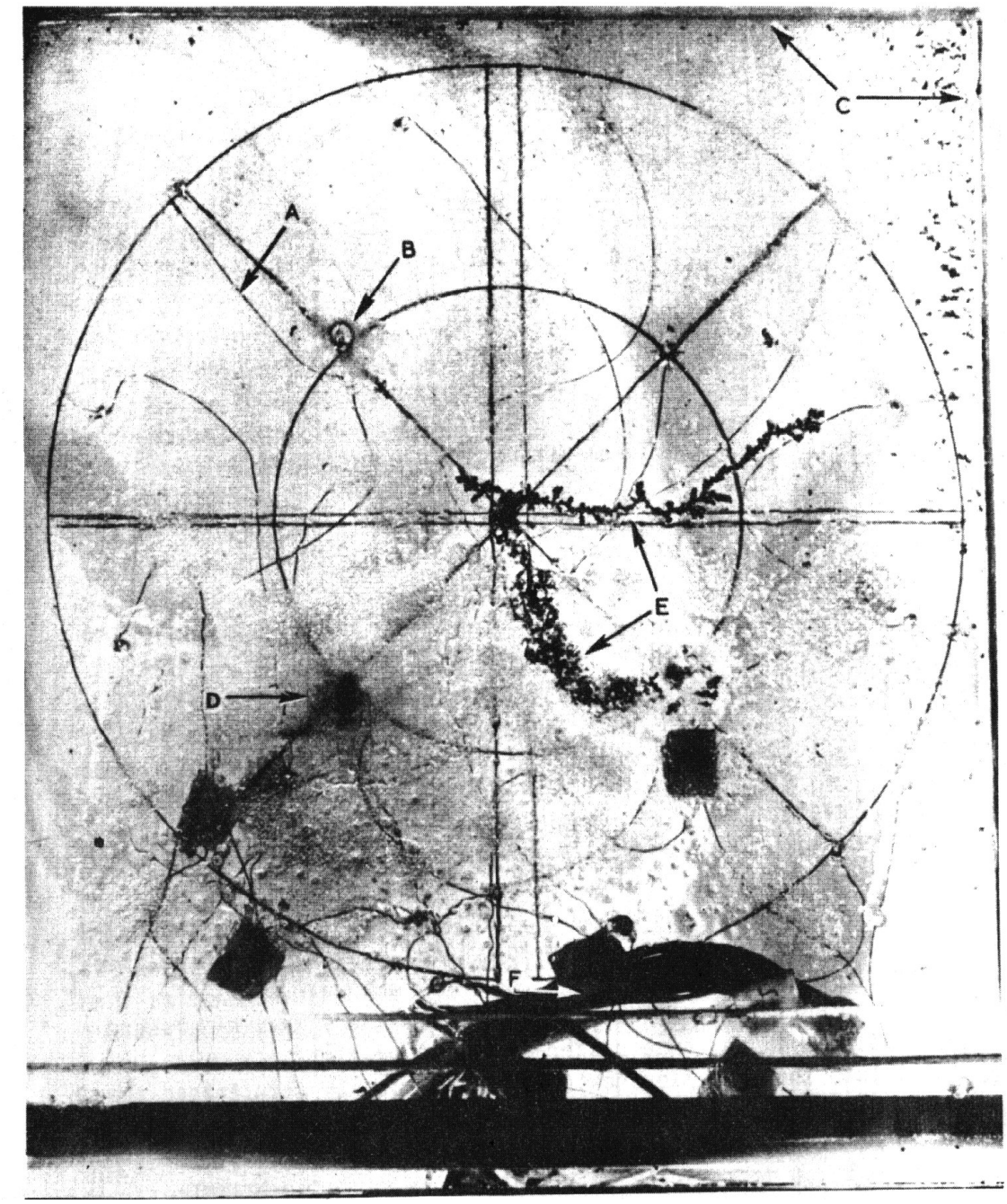


Figure 2.1: Photograph of Pask's electrochemical assemblage from (Pask, 1958b; Cariani, 1993). The black-and-white photo is of the electrochemical bath, where tree-like threads of iron (E) grow in acidic solution from electrodes (perpendicular to the page). The circular wires are a support frame. The electric field at the electrodes modifies the growth of threads, while the growth of metallic threads modifies these electric fields. By rewarding (with increased growth) the response of wire structures to particular multimodal stimuli, Pask was able to demonstrate the assembly of a wire ear which "heard" particular sounds.

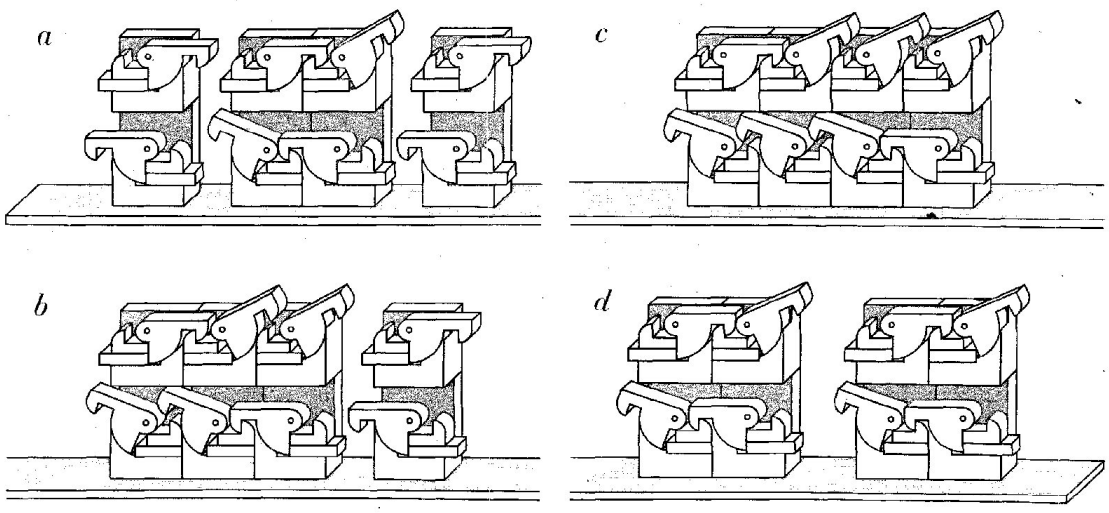


Figure 2.2: Diagram of self-replicating blocks from (Penrose, 1959). The vertical block clasp assemblies are capable of basic assembly and replication if one considers the two-block assembly of (a) an assembled object. After joining consecutively with a block on the left (b) and right (c) (assuming stochastic shaking motion), a pair of two-block assemblies is created. The two-block assemblies effectively catalyze their own production.

macroscopic devices. The search for simple discrete assemblers over unlimited scale in this thesis was partially inspired by Pask's approach.

A second major study of real but artificial assembling devices was the 1959 work by Lionel Penrose and his son Roger on systems of tumbling plywood blocks (Penrose, 1959). Perhaps Roger Penrose is better known for the beautiful aperiodic tilings which bear his name, though this work was similarly groundbreaking. By shaping blocks with unpowered mechanical latches and shaking them in a flat box, the Penroses were able to assemble larger aggregates and demonstrate shapes which built copies of themselves (Figure 2.2). The idea of physical shape directly influencing function is today an active area of research in the field of conformational switching (Saitou, 1999; Freitas & Merkle, 2004). Protein chemistry, in particular, is largely based on conformations. One canonical example is bacteriophage assembly, where parts of virii are built from self-assembling proteins (Casjens & King, 1975; Thompson & Goel, 1988). More recent chemical computing using DNA exploits nucleotide pairing to literally build computational solutions (in both senses) from DNA-encoded directives (Adleman, 1994) (discussed later in Section 2.3.2).

2.2.1 Artificial life and robotics

Physical and mathematical approaches to assembly intersected in the early 1990s with parallel research into models of biological, chemical, and robotic processes. Though mathematical models of these systems have been in constant development for hundreds of years, the widespread availability of huge computing power allowed researchers, for the first time, to simulate hundreds and thousands of interacting components. This new freedom facilitated a new type of investigation into chemistry and biology: simulated experiments on complex living systems not as they were but *as they could be*. The new direction was widely recognized as the new fields of artificial life (ALife) (Langton, 1989) and artificial chemistry (McCaskill, 1988; Fontana, 1992) in the early

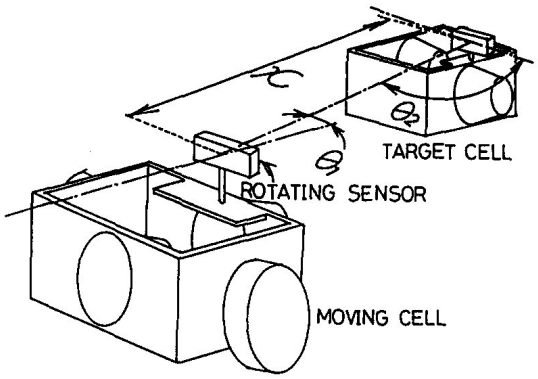


Figure 2.3: Diagram of a CEBOT module from (Fukuda et al., 1991). Multiple mobile cells search and link to one another via two complimentary ports.

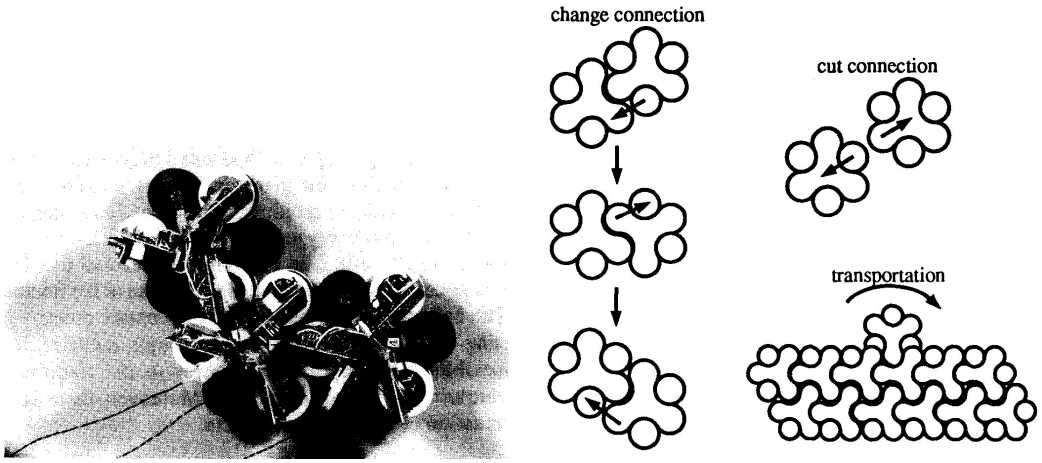


Figure 2.4: Images of fractum modules from (Murata et al., 1994). The modules form a hexagonal grid (2.4b) and move via magnetic changes on a powered surface.

1990s. Abstracted chemical and biological systems, essentially huge interacting baths of similar units, provide the second foundation to assembly research as it is conducted today. The goal of this type of research is twofold, and not only includes a biological understanding of why life processes exist in the forms they do, but also an algorithmic understanding of life processes which are difficult to mimic using artificial technology (Bedau et al., 2000).

Alongside these developments, the maturation of robotics technology led to the first macroscopic examples of active assembling devices. The core engineering ideas behind self-replicating (and self-assembling) robots began in the early 1980s with designs for replicating space probes (Freitas, 1980; Merkle, 1992; Sipper, 1998). By the late 1980s the first real devices were built, including both the CEBOT (Figure 2.3), a robot formed from individual cells echoing Von Neumann's original kinematic assemblers (Fukuda & Nakagawa, 1987; Fukuda et al., 1991), and followed closely by the fractum (Figure 2.4), a two-dimensional, hexagonal robotic lattice built from identical parts reminiscent of a cellular automata (Murata et al., 1994). These were the first prototypes of robots which complete tasks by linking together into assemblies, known today as the field

of modular robotics. The potential of robots which do not explicitly assemble but instead work in groups, swarm robotics, began much earlier in the mid-1940s but only gained momentum at a similar time (Dorf, 1990; Cao et al., 1997), though it is important to emphasize again that biological models of swarming long preceded robotic studies. As opposed to modular robots, robots in swarms do not necessarily physically attach to one another. However, from a larger perspective, swarm robotics may be considered a special case of modular robotics (or vice versa) in which units can be arbitrarily reconfigured with respect to one another. As with artificial chemistries and artificial life, the boundaries of these fields are fluid and still being explored. The reader is also referred to (Freitas & Merkle, 2004) as an excellent review of physical instances of assembling systems, though with a focus on replication.

2.2.2 The modern synthesis

There is significant overlap in modeling and implementation between varieties of cellular automata (CA), chemical assemblers, conformationally switching devices, artificial chemistries, and simulated assembling robots. For historical reasons, cellular automata work tends to emphasize the design of the rules inside the units, while artificial chemistry and conformational switching regard the interactions between units as the more important aspect. Additionally, the interactions in CAs tend to be discrete and synchronous, while artificial chemistries often investigate types of asynchronous, stochastic interactions and robotic interactions tend to be asynchronous and deterministic, but more complex. There are also many exceptions to these generalizations. In the spirit of artificial chemistry, swarm and modular robotics, and assembly reviews which compare studies along different research axes (Dudek et al., 1996; Iocchi et al., 2001; Dittrich et al., 2001; Freitas & Merkle, 2004; Bayindir & Şahin, 2007), a partial listing of the assumptions of canonical models of assembly in various fields is shown below in Tables 2.1-2.3. The comparison of assembly models is not intended as definitive, rather, the various features are meant to illustrate the various ways assembly, once disembodied from implementation, has been modeled in these disparate areas. Because of the extreme interdisciplinary nature of assembly research, models in different fields have emphasized different assembly properties. The CORAL model is also included in these tables to show how the choices made compare with other paradigms, though the full model is introduced only in Chapter 3.

Type of Model	Synchronized interactions	Stochastic interactions	Discrete interactions
Cellular Automata	yes	no	yes
Artificial Chemistry	no	yes	maybe
Conformational Switching	no	yes	yes
Swarm Robotics	no	maybe	no
Modular Robotics	no	no	yes
Chemical Computing	no	yes	maybe
CORAL Model	no	yes	yes

Table 2.1: Interaction features of different types of assembly models. Interaction refers to the way in which individual units pass information between one another.

Type of Model	Identical units	Stochastic units	Discrete units	Finite state units
Cellular Automata	yes	no	yes	yes
Artificial Chemistry	maybe	no	yes	yes
Conformational Switching	maybe	no	yes	yes
Swarm Robotics	maybe	no	maybe	no
Modular Robotics	yes	no	yes	no
Chemical Computing	maybe	yes	maybe	yes
CORAL Model	yes	no	yes	yes

Table 2.2: Individual unit features of different types of assembly models.

Type of Model	External direction	Full program-per-unit
Cellular Automata	no	no
Artificial Chemistry	no	no
Conformational Switching	no	no
Swarm Robotics	yes	yes
Modular Robotics	yes	yes
Chemical Computing	yes	no
CORAL Model	yes	no

Table 2.3: Control features of different types of assembly models.

It is important to emphasize again that there are many exceptions to the yes/no declarations above.

Table 2.1 compares different assembly models on criteria derived from the way units interact with one another. Models using synchronized interactions require all units to change state at the same time. Stochastic interactions means that the interactions are chosen randomly, or have some sort of random factor affecting them. Discrete interactions are interactions which happen in an all-or-none fashion, either units are chosen to have a particular interaction or they do not.

In contrast, Table 2.2 compares the way units themselves are modeled. Units may all begin as identical, or there may be many different kinds of units used to populate the simulation. Like stochastic interactions, units themselves may operate stochastically and use (semi-)random state changes. As shown in the third column, units and assemblies may be discrete objects, or there may be ambiguity in defining boundaries or to what other parts a unit is attached. Finally, assembling units may be modeled as infinite-memory general-purpose computers, or strictly limited in state.

The final Table 2.3 highlights the way in which these various assembly models are used or controlled. Units may be subject to external influences during the assembly process, or require some external direction in order to be used. Related to this idea is whether each individual unit is itself programmed with the full instructions, or whether the units are individually executing simpler programs which result in the desired function in combination.

Some interesting general properties become clear when looking at the above tables. To begin, it somewhat justifies the "object" definition of assembling systems given in Chapter 1, in that the main properties shared at least partially by all of the above models are discrete, identical units. As an aside, this is not to say that all assembling systems use exactly identical units, but only

that with discrete units it is often possible to postulate an uber-unit which subsumes all the unit types if multiple types of unit are required, enabled initially for a single type. The main points of divergence between models seem to be along the stochastic interaction, external direction, finite state unit, and program-per-unit axes. While there is largely consensus on the other factors, any model investigating assembly in general must make these choices, potentially limiting the areas to which the model will be applicable.

Because it was designed to investigate realizable assembling units but ignore particular interaction details, the CORAL model described in this paper combines the strict interactions of modular robotics with stochastic interactions and simple units of artificial chemistries. This enables both structural realism and huge scales. The highly-programmable and high-memory units of swarm robotics and modular robotics are possible due to the huge scale difference between microelectronics and macrorobotics, but this difference does not exist in other domains and is reduced as robotics shrink. Assuming finite state units is a safer choice, and such units are generally much easier to build and predict even in macroscopic devices. Along the same lines, it is deemed highly important that external direction over assemblies be possible. For reasons that will be elaborated further on in this chapter, program-per-unit methods tend to be fundamentally less flexible in reconfiguration than approaches that explicitly recognize the role of external influences.

The misleading historical distinctions in which each of these models developed have recently been falling away, leading to a more general appreciation *self-constructing, open, far-from-equilibrium systems* (Kauffman, 1993) and the more concise notions of *dynamic self-assembly* (Whitesides, 2002) and *constructive dynamical systems* (Fontana & Buss, 1994; Banzhaf, 2004). As defined by Fontana and Buss:

> We develop an approach where no particular fixed network is known *a priori*, by considering a finite ensemble in which interactions among objects repeatedly construct new objects... A constructive dynamical system is, therefore, characterized by two components: a dynamics in phase space and a dynamics of the system's support.

The study of constructive dynamical systems motivates the previously-mentioned definition of assembly, and again emphasizes treating assembling systems as systems of distinguishable objects. The study of all general constructive dynamical systems is an exciting topic beyond the scope of this thesis, and so here we have limited our review of these works to those explicitly designed to create, model, or measure the production of new entities at multiple scales. In the next section, the more recent developments in this and other contexts will be presented.

2.3 Current work on assembling systems

In a seminal Artificial Life paper, a set of 14 open questions were posed by leading researchers as important challenges remaining in the field (Bedau et al., 2000). Question 8 of 14 is a challenge to "create a formal framework for synthesizing dynamical hierarchies at all scales." Assembly is mentioned explicitly as a core component needed to build these processes, in multiple contexts, along with a formal framework in which to model constructive systems. The work in this thesis directly addresses this challenge from both the modeling and implementation perspective. In the past 10 years, much progress has been made on this topic, (Lenaerts et al., 2005), but the question

remains unanswered (Bedau, 2007) and addressing it fully would unlock new scalable technology and better understanding of multi-level networks in biology, chemistry, and computer science.

As might be expected from the diversity of viewpoints on assembly, work toward building and modeling systems exhibiting order at multiple scales comes from a variety of directions. Dynamic hierarchies have been defined using philosophical, computational, and informational measures, often tied to particular implementations. Related measures of complexity sometimes address scaling measures as well. There have in parallel been many less formal approaches simulating and building various types of assembling systems with interesting multi-level behaviour. A survey of this research and the current progress made toward understanding dynamic hierarchies and multi-scale behavior is presented below.

2.3.1 Modeling dynamical hierarchies

Many approaches to dynamical hierarchies have tended to focus exclusively on the problem of formally identifying multi-scale structure, and several competing paradigms exist to describe the idea in various contexts. As early as 1969 scientists acknowledged the mathematical study of hierarchical systems was valuable (Mesarović & Macke, 1969), but later work by Baas (Baas, 1994) popularized the ideas for artificial life audiences and proposed a fairly similar set of properties required for a system to generate new properties from interacting parts at multiple levels. These properties are inherently tied to the notion of an *observer.* A number of initial (level 1) objects are required as a given, which may be grouped into different indexed sets $S^1_x =, x \in \mathbb{N}$. An interaction function $I^1(S^1_x)$ and observation function $O^1(S^1_x)$ are defined over these sets, intuitively capturing the ways in which the objects can interact and the ways in which they can be viewed. A constructive grouping or result function R then takes these observations and creates a second-order structure using some process, often representing ideas like equilibrium structure (though other functions are not disallowed). Second-order structure is the result of this function applied to the first order structures:

$$S^2_y = R(S^1_x, O^1, I^1) \text{ where } x, y \in \mathbb{N}$$

A higher level property or dynamic P is created when:

$$P \in O^2(S^2_y) \text{ and } P \notin O^2(S^1_x)$$

O^2 is an observation function for higher-level structures which may or may not be the same as O^1. This hierarchy of organization can be extended upward indefinitely to any level, each new level potentially requiring a new observation function or interactions. (The above is a summary from (Rasmussen et al., 2001b) and (Baas et al., 2004).)

While Baas's model incorporates core ideas of hierarchical organization, it is not very precise because the notion of observation is not very strict. Trivial hierarchies can be created, e.g. solely aggregating cellular automata (Dorin & McCormack, 2002), and others have proposed extensions and new models to formalize hierarchy in agent-based (Groß & Lenaerts, 2003), information theory (McGregor & Fernando, 2005), or compatibility (Rowe et al., 2005) terms. Further approaches, taking insights from models of complex systems in physics and dynamical systems

theory have been introduced by Jacobi (Jacobi, 2005) and in the form of Chemical Organisation Theory (COT) (di Fenizio et al., 2000; Dittrich & di Fenizio, 2007).

Other researchers have attempted to capture hierarchical organization using models from computer science. These formalisms have the advantage of depending only on the system interactions themselves for detecting complexity, and are easily analyzed mathematically. It is possible to decompose a given finite state automata into a nested series of sub-automata using the Krohn-Rhodes decomposition, which can then be used as a bias-free measure of the hierarchical complexity of a system (Nehaniv & Rhodes, 2000; Egri-Nagy & Nehaniv, 2004, 2008). A more constructive model using the complexity of interactions in a finitary process soup (FP soup) has also been proposed, which simulates assembling systems as interacting state machines (Crutchfield & Young, 1989; Crutchfield & Görnerup, 2006). The complexity measure for the atomic ε-machine units (fully-connected, deterministic finite state transducers) can also be applied to population interactions between units, allowing meta-unit organizations to be easily tracked.

While these numerous measures of dynamical hierarchy provide different perspectives and "slices" of how the system is organized, no broad consensus seems to have emerged privileging one over the other. Arguably, this is because these various hierarchical decompositions are difficult to apply to many realistic simulations or data which would weed out measures which do not have much relevance to actual systems. (An exception to this, however, is chemical organisation theory, which has been used as a framework for chemical computing (Matsumaru et al., 2005).) An additional factor is that the measures themselves are not constructive, nor are they intended to be. To build novel types of many-level dynamical hierarchies requires other methods, and so ease of construction or the interesting behavior generated cannot really be used as a criteria. To those who wish to define hierarchy rigorously, such work must take the existence of hierarchies as a given. Perhaps as a consequence, however, no mathematical search for automata or primitives which hierarchically assemble indefinitely has been completed thus far, using any of these models, though such automata may exist. Instead, artificial chemistries are the more-or-less concrete experiments into *how* structure can be generated.

2.3.2 Dynamical hierarchies in artificial chemistries

Artificial chemistries provide a useful middle ground between pure theoretical investigation into multi-scale systems and the types of interactions needed to realize them. Many varieties exist, often involving some sort of assembly, while all tending to share the basic concepts of many individual parts interacting using predefined rules to form larger organizations (Dittrich et al., 2001; Banzhaf, 2004). Probably the canonical example of these ideas is the previously-mentioned AlChemy artificial chemistry simulation of small λ-calculus functions (Fontana & Buss, 1994, 1996). By transforming one another and being transformed, at least three levels of program interactions can be discerned. The programs themselves undergo interactions with one another which are only in some cases directly mappable to physical space, though it is mentioned that functional families can be generated by structural families of related chemicals. The previously mentioned FP-soup model (Crutchfield & Görnerup, 2006; Görnerup & Crutchfield, 2008), when divorced from the complexity measure, contains a similar idea using finite state machines (FSMs), as does newer work in reflexive FSMs (Salzberg, 2007). Other tape-based models draw inspiration from

the Turing machine model of computation (McCaskill, 1988; Dittrich & Banzhaf, 1998; Ikegami, 1999). In these, binary strings act upon one another as alternately program or data, which is hypothesized to be a superset of the types of manipulations chemical structures perform on one another. Single or multiple organizations of dynamically stable units have been observed using these approaches, though again the interactions of derived units are not related in any simple way to a physical interpretation of the ancestors. Similar to these ideas are simulations of autocatalytic protein sets (Kauffman, 1993) or metabolism (Bagley et al., 1991), which demonstrate the self-organization and metadynamics of cyclic organizations built through catalytic closure, i.e. hypercycles (Eigen & Schuster, 1977). Approaches using traditional cellular automata and the organizing of higher order groupings of cells has also been demonstrated (Baas & Helvik, 2005; Helvik, 2005; Hoekstra et al., 2007), with applications to other formal distributed models. These models assume *a priori* behavior at each scale exists, however, and do not create organizations themselves.

In general, these computational artificial chemistries have been shown to be a rich medium in which to generate multiple levels of interactions and sometimes structure (often, for example, exhibiting self-replication). The CORAL simulation environment, itself a close relative of the FSM soup approach, builds upon their previous success and demonstrates that such soups are capable of unlimited assembly. Thus far, only limited examples of structural or organizational hierarchies had been generated using these approaches, with most of the work in organizational hierarchy. A drawback to purely computational approaches, however, is that they lack physical realism without additional constraints, and these constraints may significantly change the allowable classes of behavior. Basic notions such as conservation of matter and unit topology have often been ignored in favor of the abstraction, and so in this light making the interactions and units of the system compatible with basic physics became an explicit goal of the CORAL model.

Other artificial chemistry research is more directly inspired by biological chemistry, which often demonstrates many levels of hierachical dynamics through assembly. An early example is one of the most interesting: (Thompson & Goel, 1988), where a novel artificial chemistry and cellular automata variant, the movable finite automata (MFA), is used to simulate the construction of a virus. The original work was limited to rectangular interactions, but was later extended to use more realistic spherical interactions (Shirayama et al., 2004). The MFA model is interesting as it incorporates the elegant finite automata model of computation with a very general notion of assembly via complementary ports, quite close to that used in the CORAL model. However, due to particular assumptions used by the model about bond strengths and bond topologies it becomes difficult to apply outside the domain of viral assembly.

Simulations of autopoeisis (Ono & Ikegami, 2001) have been shown to demonstrate the generation of meta-structures and metabolism from simple parts, as have other grid (Mayer & Rasmussen, 1998; Banzhaf et al., 1999) and non-grid-based (Dorin, 2000; Hutton, 2002, 2007) approaches inspired by biological mechanisms. In the latter work by Hutton, using rules loosely patterned after the actual cellular mechanisms, proto-cell type patterns have been built which require at least three hierarchical assembly processes. The model of (Doursat, 2008) also demonstrates multi-scale constructs, using an "excitable canvas" of mobile cells with modular genetic regulatory networks (similar to amorphous computing (Abelson et al., 2000; Nagpal et al., 2003)). Though

no explicit measure is made, such an approach in theory could generate organizations at many scales at the cost of increasing complexity in the core units. The JohnnyVon artificial chemistry is based on a biological metaphor of nucleotides, and attempts to use the pairing functionality to build much larger regular structures (Smith et al., 2003; Ewaschuk & Turney, 2006). These static assembled structures can be quite impressively large, but the particular pairing force-field model holding assembled structures together is again difficult to apply generally. Multi-set based self-assembly of cell-like computing structures (membrane or P-systems (Păun & Rozenberg, 2002)) is another recent development (Bernardini et al., 2005, 2007), though in this particular work the cells are taken as a basic unit and multi-scale structure is not explicitly addressed. Membrane systems naturally have a multiply-nested approach, and are similar in structure to Baas's hypercycles (Baas, 1994) or bigraphs (Milner, 2009).

"Real" artificial chemistry

In a broader sense, all of chemistry (and perhaps physics) can be considered assembling systems, and significant progress has been made simulating realistic assembling chemical systems (Klein & Shinoda, 2008) in particular areas. The goal of these studies is somewhat different from the minimal approach demonstrated here and in other artificial chemistry studies. Realistic models are useful for prediction and understanding of real chemical systems, whereas artificial chemistry models such as the one presented here seek to understand the core functionality needed for particular behaviors. For example, it is difficult using realistic models to analyze and describe what general properties, if any, allow simple amino acids to perform so many functions when used in combination with one another. Is it an accident of evolution that (more or less) 22 are used, and might they be even more powerful if more were added or the chemical behavior was different? These are important questions when attempting to mimic natural assembly using our own artificial constructions.

However, a new synthesis of chemistry and computer science is again underway, echoing the merge which created artificial chemistry in the 1990s. Currently described as *rule-based modeling* (Hlavacek et al., 2006), the idea is again to integrate computer science with biological processes but, this time, with vastly greater computing power and knowledge of detailed cellular processes. The κ model (Danos & Laneve, 2004) and similar precursor models such as BioNetGen (Faeder et al., 2005) are powerful process calculi and agent-based approaches (respectively) applied to real chemical processes currently difficult to model in entirety. In particular, the combinatorial explosion caused by assembly poses problems for conventional approaches (Danos et al., 2007). The κ model uses notions of proteins, binding sites and interfaces quite close to the units, ports and state of the CORAL model and also to robotic self-assembly using graph theory via Klavins (Klavins et al., 2006b; Danos & Tarissan, 2007) and the π-calculus approach of (Regev et al., 2001; Regev & Shapiro, 2002). The major difference between these approaches and the CORAL model is that the CORAL model uses internal state to enforce local operations, whereas local operations are constraints on the syntax of κ reactions and graph-theoretic assembly (Figure 2.5). It is a deep question, not fully answered to date, how to "bunch" local interactions together into higher semantic levels, though rule-based approaches show promise (Danos et al., 2008). The recursive assembly discussed throughout this thesis is one form of such bunching, applicable to particular kinds of interactions.

The κ molecular process model

It is particularly useful to illustrate in more detail the workings of the κ process model, as it is a modern fusion of ideas from other successful biological process models and a "purer" computational approach. Again, the goal of the work differs from our own, but the closeness in design of formal molecular models and many parts of the CORAL assembling system leads to many similar challenges.

The original intention of the κ-model was to model protein interactions (Danos & Laneve, 2004), and as such places emphasis on topology changes between and within linkable units. Prespecified rules changing the units' port or *site* topology in response to previous changes have the power to emulate the high-level behavior of active sites in proteins. No physical environment is specified explicitly; one can visualize the κ model as a well-mixed reactor with anonymous units of different types, where each unit instance has a particular set of open sites. To be more specific (much of the following details are derived from (Danos & Laneve, 2004; Danos et al., 2006; Danos & Tarissan, 2007)), the basis of the model is:

- a countable set of protein names $\mathcal{P} \in \{A, B, C, \ldots\}$
- a countable set of edge names $\mathcal{E} \in \{x, y, z, \ldots\}$
- for all $A \in \mathcal{P}$, a number of *sites* of A written (A, i), where $i \in \mathbb{N}$ and $i < \mathfrak{s}(A)$
- a signature map $\mathfrak{s} : \mathcal{P} \to \mathbb{N}$, defining the maximum number of sites of a protein name A

The purpose of the edge and protein names is fairly straightforward. The *sites* of a particular protein name are numbered and limited to some maximum defined by $\mathfrak{s}$. A *solution* in the κ model is an algebraic construct with the following recursive syntax:

$$S := 0 | S, S | A(\rho) | (x)(S)$$

The values 0 and S, S allow one to describe an empty solution or a solution of many proteins, while the $A(\rho)$ indicates a protein name $\in \mathcal{P}$ with *interface* ρ. Intuitively, this defines solutions as collections of named proteins, each instance of which has a particular set of ports opened, closed, or connected. The final term is a *new* operator, which binds the (edge) name x in the adjacent solution. (Note that in this syntax the protein name A refers to a particular *type* of protein, whereas the name x refers to a particular edge *instance*.) Generally this binding is used to introduce shared edges between proteins modeled in a solution, where x is a bond name used in the protein interface ρ. The syntactic form is extremely similar to that of the π-calculus, by design.

What remains to be described is the protein interface - ρ is a partial map $\mathbb{N} \to \mathcal{E} \times \{h, v\}$, where each integer in the domain of ρ can be thought of as corresponding to a site of a particular protein name. As described above, this interface defines the state of the sites of the protein, with each site either connected via a named edge, hidden (h), visible (v), or not present (as is allowed by the partial map). Interfaces are much easier to understand using a graphical notation, where $i + \bar{j} + k^x$ indicates a visible site i, a hidden site j, and a site k connected to edge x.

κ solutions and reactions

The syntactic state of a chemical solution is intended to model the topological state of a real protein solution. Chemical reactions dictate how bonds form and are broken in real solutions, and corresponding *reactions* in the κ model dictate transitions between syntactic forms. For each chemical system modeled, the reaction set $\mathfrak{R}$ contains mappings between solution "prototypes." A particular solution matches one of the prototypes in the domain of the reaction map if it is syntactically identical up to edge renaming (though there are some additional subtleties with commutative terms). As an example, Equations 2.1 describe reaction rules corresponding to the topological reactions diagrammed in Figure 2.5a. These reactions are, respectively, the *activation* of particular sites and the linking or *complexation* of units via the active sites.

$$\begin{aligned} A(\bar{a}+i+j),B(b+h),C(c+k+\overline{c'}+c'') &\rightarrow (e)(A(a+\bar{i}+\bar{j}),B(b^e+h),C(c^e+\bar{k}+c'+\overline{c''})) \\ A(\bar{a}+i+j),B(b+h),C(c+k+\overline{c'}+c'') &\rightarrow \\ & (e)(f)(g)(A(a+i^f+j^g),B(b^e+h^f),C(c^e+k^g+c'+\overline{c''})) \end{aligned} \tag{2.1}$$

Sets of reaction functions define a transition system over κ solutions: iterating from a solution s_i, one can progressively match sub-solutions to inputs in the reaction rules and replace them with instances of reaction-rule outputs in a new solution s_{i+1}. This matching process is immediately analgous to the subgraph isomorphism used for matching in the graphical reaction rules of (Klavins et al., 2006b), for example, and graph grammars in general.

Comparison between the CORAL and κ models

Importantly, the syntax of κ solutions is broad enough to encompass models with no simple physical analog; for example, solutions with a single edge between 20 different proteins. In addition, reaction rules are not constrained to operate within any particular size bounds, making non-local interactions possible. Several restrictions on the allowed reaction forms are required so that physically plausible models remain so, but it is up to the model user to verify that the reaction rules are sufficiently local (properties also shared by graph grammar assembly models (Danos & Tarissan, 2007)). This seems at first an essential difference between these and the CORAL model, since in the CORAL model only unit-local interactions or interactions between directly linked units can be represented. As shown by (Danos & Laneve, 2004), however, a pairwise-restricted version of the κ model, the $m\kappa$ model, can emulate every valid κ reaction. This emulation generally requires additional state, however, depending on the high-level operation.

As one goal when developing the CORAL model was to investigate the minimum state required scale-invariant self-assembly (for definitions of scale-invariant explored throughout the remainder of this thesis), this emulation step is of primary interest. Pairwise restrictions ensure that in writing down the reaction rules one cannot conceal steps which require additional state be added for synchronization, since then recursive assembly algorithms may be written which require unlimited state. This is a feature in the κ model which gives power to the syntax, however it is inappropriate when investigating minimal units.

Another notable difference between the models is in the nature of combined operations. While the solution network state is contained in the links defined on each protein in the κ model, in the CORAL model a barrier is drawn between the internal state of units and the environmental state

of links between these units. The separation is due again to the goal of the work - the state of CORAL units is meant to represent the designed-in complexity (and should be minimized), whilst environmental complexity emerges through assembly interactions (and should be maximized). The second major difference, related again to the modeled environment, is the addition of a single broadcast signal in the CORAL model. To date, κ and graph-grammar models have been used to investigate *closed* environments, where a target is specified and the reaction rules derived for that target. As explained in more detail below, this fundamentally limits the avenues available for investigation, in particular relating to structures which act dynamically.

Though 3 introduces the full CORAL model, Figure 2.5 illustrates the overall similarities in graphical construction to the κ-model. As described above, the state or interfaces of κ proteins are contained in named site mappings (drawn as small external circles), while the state of CORAL units is defined by patterns of tokens in the state of internal Petri net *places* (drawn as small internal circles). Transition rules in the κ model are not pictured, as they are defined externally via reaction rules, but transitions are included in CORAL model diagrams as rectangular *transitions* linking the places.

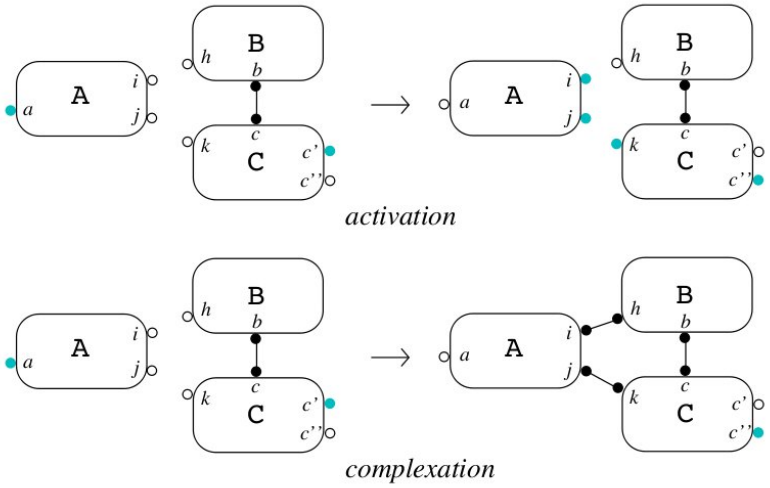
Further chemistry-inspired models

The previously-mentioned Chemical Organisation Theory (di Fenizio et al., 2000; Dittrich & di Fenizio, 2007) works at the higher level of chemical kinetics, but shows similar promise in modeling poorly understood reaction networks. Given a set of reaction rules, *organisations* can be defined as self-maintaining subsets of chemical compounds. Organisations are tied deeply with dynamical analysis of the system, and any sufficiently complex chemical system generates hierarchies of these organisations. No structural information is needed or provided by this view, but when structure is largely ignorable (such as in chemical computing) it provides a powerful interface into complex reaction networks and hierarchical decomposition of chemical behavior (Benkö et al., 2009).

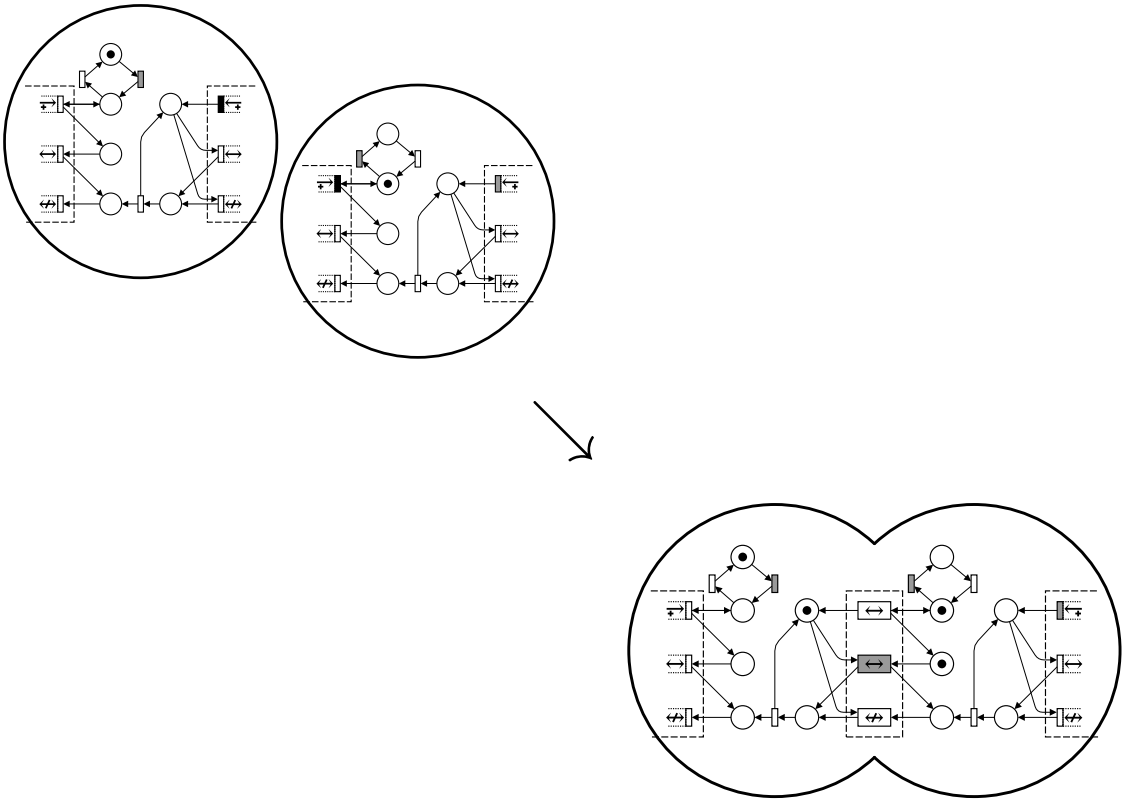
Parallel biochemical work modeling the assembly of real DNA (Fontana, 2006), a molecule with highly discretized interactions and combinatorial chemistry, has also led to new formulations of self-assembly theory using only tiles with complementary edges (Adleman, 2000; Seelig et al., 2006). Nanoscale versions of these tiles have been formulated using DNA base-pair binding to enforce complementarity, resulting in impressive constructions such as DNA Sierpinski triangles and tesselated shapes (Rothemund et al., 2004; Rothemund, 2006). Dynamic multi-level and modular constructions are not explicitly investigated, though the fractal forms are quite suggestive (Figure 2.6). The tile model significantly differs from the CORAL and above rule-based assembly models, however, in that it assumes a combinatorial number of complementary "glues" can exist between tiles. This may be practical for long strands of DNA and robotic devices, but it limits the applicability in devices without an initially large state or combinatorial number of types.

2.3.3 Multi-scale robotic assembly

Research in robotics now finds itself facing problems which were formerly confined to the biological and chemical worlds. As designs move toward distributed components which are cheaper and potentially more flexible, understanding the behavior of arbitrary assembled structures requires fundamentally new paradigms. This is similar to the situation in biochemistry, but with an added



(a)



(b)

Figure 2.5: Comparison of edge creation / linking in the κ-model (2.5a) (Danos & Laneve, 2004) and the CORAL model (2.5b). State is contained in port activations on *proteins* in the κ-model, which may be hidden (blue circles), open (white circles), or linked (black circles), and linking takes place via rules which modify these ports and edges. As will be seen in Chapter 3, state is internal in the CORAL model but drives ports which may be disabled (white rectangles), holding (black rectangles), or connected (merged rectangles). The reaction rules in the κ-model are specified explicitly, while in the CORAL model the rules may be partially implicit with intermediate computations. Both models are equivalently expressive, and one can translate CORAL units into the proteins and reaction rules of the *m*κ-model (a restricted pairwise version of equivalent power) or vice-versa.

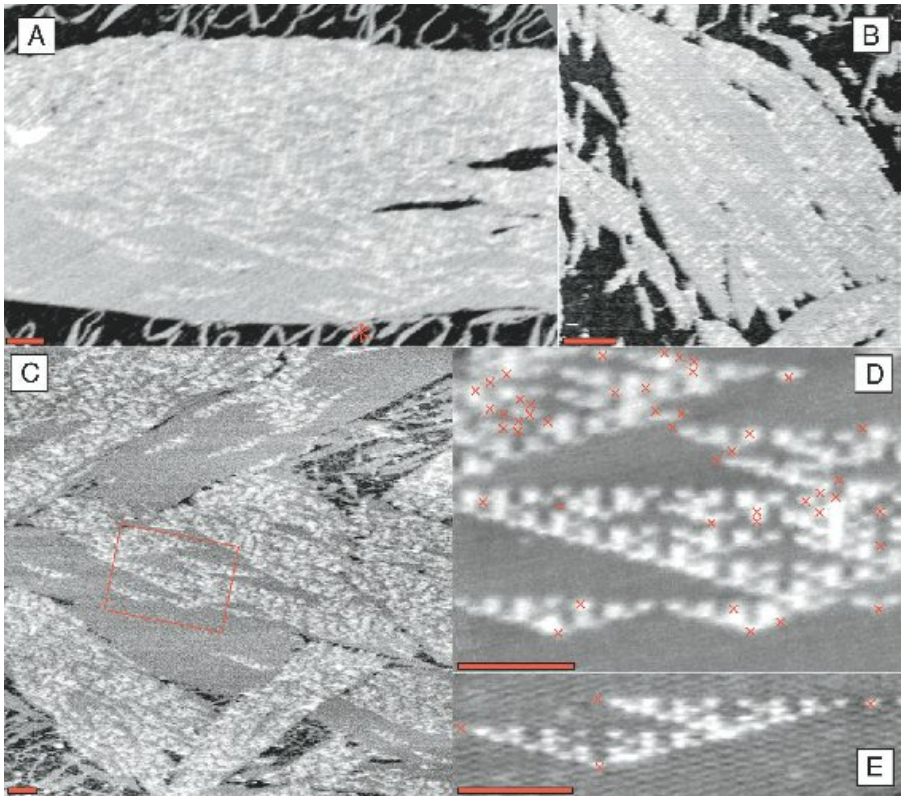


Figure 2.6: Images of DNA assemblers from (Rothemund et al., 2004). Assembled Sierpinski structure (with errors) can be seen in (D) and (E). The patterns were programmed from DNA tiles with edges of particular specificity, so that tiles representing 1-values may only attach to two previous tiles with values (0,1) or (1,0).

twist - the units themselves can be built to *any imaginable form*, and the object is not to observe but to *control*. The question is not so much if a particular robotic system can be built, but rather how to build the system most effectively and what kind of primitive devices to use as the base. To echo the previously-posed question: it is as if one wished to design a new set of amino acids. The field is still in rapid development, however, and though these issues are widely acknowledged most approaches to date have tended toward custom implementations for particular platforms, with rare exceptions.

Because of the complexity inherent in building an assembling mechanical system, work in assembling robotics has only recently concerned itself with the assembly of large (thousands or more) numbers of parts (Yim et al., 2007a, 2009). To a large extent, research in this area is bifurcated into swarm and modular robotics. Swarm robotics research primarily investigates the designs and protocols of distinct units which primarily communicate with one another, while modular robotics investigates robotic units which can physically connect with one another. There is overlap between the two areas, such as the original mobile CEBOTs (Fukuda et al., 1991), the Swarm-bots project of multiple mobile robots with attachment grippers (Tuci et al., 2005), or the mobile Jasmine robots which dock together to form robot structures (Kornienko et al., 2007).

Reconfiguration and control

One shared problem in both these areas is known as the "configuration problem": given a number of robotic units, how can one direct them into creating a desired formation or structure. There have been many practical (see review (Ota, 2006)) and theoretical studies of this problem (see review (Prencipe & Santoro, 2006)) in swarm robotics, where it is sometimes called the (arbitrary) pattern formation problem (Suzuki & Yamashita, 1996; Bahçeci et al., 2003; Flocchine et al., 2008), formation control and reconfiguration (Gazi & Fidan, 2006), or more generally just self-organization. In modular robotics it is more often called the (self-)(re-)configuration problem (Chen & Burdick, 1995; Dudek et al., 1996; Parker, 2000; Yim et al., 2007a). Modular robotic studies often go further into self-assembly, as task-related reconfiguration is the central goal of their study.

Reconfiguration results often depend heavily on the type of communication allowed and the ways in which the robots can move and sense. Broadly, these can be divided into centralized and decentralized approaches (Dudek et al., 1996; Cao et al., 1997; Iocchi et al., 2001; Bahçeci et al., 2003). Centralized approaches are typified by a single controller entity sending individual robots direct commands at each time interval. Decentralized control implies each robot makes decisions based largely on local information from attached sensors. For large numbers of robots, there is general agreement that the centralized approach is more computationally intensive, less flexible, and less scalable than decentralized control (Dudek et al., 1996; Bayindir & Şahin, 2007), but it is not fully understood how decentralized robots can be designed to make complex decisions and structures (particularly dynamic structures) (Whitesides, 2002; Bahçeci et al., 2003). Swarm robotic work often takes a partially centralized approach, while most, if not all, modular robotic devices use decentralized networking.

2.4 A middle way

The work presented in this thesis points at an intermediate path. The difficulty of controlling huge numbers of units (e.g. robots, agents, or molecules) from a central source does not primarily arise from broadcast communication issues, but instead from the computational and bandwidth load of addressing individual robots. At all scales, broadcast signals can often be designed simply and cheaply, either by flooding an environment with particular chemicals (Matsumaru et al., 2005; Rothemund, 2006; Wu et al., 2009), electrical signals (Pask, 1958a; Murata et al., 1994; Hamad-Schifferli et al., 2002; Kirby et al., 2007), (ultra)sound (Merkle, 1992), radio waves (Fredslund & Mataric, 2002; Murata et al., 2004; Kornienko et al., 2007), heat (Livstone et al., 2006), or light (Yim et al., 2007b). Wireless communication is not often used for broadcast in prototype modular robotic systems (since reprogramming through direct-wiring is simpler during development), but ultrasound and light sensors are used in the control of many modular robotic units (Fukuda et al., 1991; Yim et al., 2000; Suh et al., 2002; Castano et al., 2002; Jørgensen et al., 2004; Bishop et al., 2005; Tuci et al., 2005; Ishiguro et al., 2006). Alternately, semi-broadcast communication can be performed using seed units or parent controllers which rely on connected units to re-transmit the signal (Kotay et al., 1998; Yim et al., 2000; Castano et al., 2002; Støy et al., 2003; Murata et al., 2004; Jørgensen et al., 2004; White et al., 2005; Zykov et al., 2007; Krishnan et al., 2007; Yim et al., 2007b; Kornienko et al., 2007), though disconnected units are not addressible in the same way. The above references are not exhaustive; sending chemical, physical, or radio-frequency signals is commonplace in many fields and the referenced works are simply examples from assembly contexts. In contrast to the broadcast approach, however, sending individualized signals to large numbers of units *is* a highly difficult task for a centralized controller. Depending on task complexity, large amounts of centralized processing may be required, and, perhaps more importantly, frequently transmitting individual unit actions is an inefficient use of communication bandwidth. In chemical or other microscopic systems, such individualized control is probably impossible to achieve by any means.

A model is proposed in the next chapter, the CORAL environment, which uses the simple aspects of broadcast signals while explicitly assuming that addressing individual units is impossible. All broadcast signals (or background signals, as they are later denoted) are received by every unit in the simulation, though they may be ignored by units in particular states. If these assembling units were used as a basis of a new type of stochastically assembling modular robot, to use a macroscopic example, they would require only a single remote control to manipulate them *all at once*. The idea is similar to that of the "conductor" robot in (Fredslund & Mataric, 2002) (or indeed, a human conductor) which sends a small, periodic broadcast indicating the type of formation the other robots should assume but not directly how to achieve it. Despite the presence of a leader robot, the algorithm is still considered a decentralized approach by (Bayindir & Şahin, 2007).

This single controller, sending simple broadcast signals to the entire environment, is able to assemble arbitrary numbers of robot-structures or unit-structures in parallel. These structures are sometimes called *devices* to emphasize their functional nature. If a single device is removed from the environment, a copy of the same controller used for assembly can also usefully control the device in isolation. The idea is a shift from pre-programming a unit or group of units with some target structure in mind, instead focusing on properties that make units (robotic or otherwise) use-

ful components in *any future (and unknown) device*. Since we (the robotic or chemical engineers) have not been given a decision about which particular device we will need to make, we want to ensure that the collection of units is able to build all sorts of different machines. Once the new device has been decided, somehow this information will have to be transmitted from some *external* source (to reprogram all the units, for example). A very simple model of this external source, chosen in the CORAL model, is a broadcast signal. Other choices are possible. Compared to other methods, broadcast signals are easy to simulate and analyze, re-use the useful broadcast control mechanism for individual assembled devices, and assume no particular inter-unit communication capabilities in the units themselves.

The paradigm shift from programming assembling units for a known task to programming units for many future unknown tasks somewhat muddies the water between centralization and decentralized algorithms. By definition, units not designed to build predetermined devices or perform predetermined tasks require some external information channel through which these may be specified. This channel may imply centralized control, but the label is misleading in the context. If, for example, the external channel allows only very small amounts of information to pass, any extra complexity in the assembled structure must be pre-specified in the units themselves. These units are decentralized in a sense, but less general assemblers and probably less useful. If this channel is wide, and essentially reprograms each unit individually as if a wire was being attached, the actual assembly process will also effectively be fully decentralized. The reprogramming step, however, ensures there is no benefit to this approach as compared to designing for known tasks. Note that current research into compiling centralized tasks (usually structural) into distributed sub-tasks for each unit also requires preprogramming and/or reprogramming the distributed instructions (Nagpal et al., 2003; Kondacs, 2003; Beal, 2005; Klavins et al., 2006a; Danos & Tarissan, 2007; Grushin & Reggia, 2008; Costelha & Lima, 2008; McNew & Klavins, 2008; Ashley-Rollman et al., 2009) (though this is a rather elegant way of doing so).

Between these two extremes lies a useful maximum, where each unit contains just enough information to be effectively reconfigured while retaining the flexibility to build many different types of devices in response to external signals. Each unit needs no individual reprogramming and is largely reactive in nature, but through broadcast environmental manipulation non-reactive, controllable structures can be formed. To contrast the reconfiguration problem, this might be phrased as the "pre-configuration" problem, and has been addressed only partially. This thesis argues that for assembling devices whose benefits primarily lie in simplicity and flexibility, the pre-configuration problem is a more appropriate target. The idea might also be viewed as a special case of compiling centralized instructions, where reprogramming happens implicitly through the new collective behavior of the structures formed.

Perhaps the best way to understand this shift is through the common biological metaphors used today in assembly research. Cells, as the smallest self-contained unit of life, have been a traditional inspiration for assembly research, e.g. cellular automata, the French flag problem (Wolpert & Dover, 1981; Miller & Banzhaf, 2003), the cellular robot (CEBOT) (Fukuda et al., 1991), and amorphous computing (Abelson et al., 2000). Cells, however, are highly nontrivial structures, with complex interactions that make them difficult to control individually and especially in combination. Each is built using incredible protein machines, however. These largely reactive

structures, when directed by chemical signals, become devices which perform a huge variety of tasks. Even better, each of these structures is made from 22 or so rather simple parts. If we wish to engineer our own assembling devices, particularly ones which need to radically and dynamically reorganize, we might do well to start down one level from cells and understand the requirements for artificial "amino acids" through which all else can be built. The CORAL model is designed for this search.

As a final clarification, there is a circular or autopoetic (Maturana & Varela, 1980) aspect to real proteins, in that the chemical signals emerge from the operation of the protein machines themselves. This feedback is undoubtedly important, though it is argued here that one must first understand the limitations of the straightforward, external controller approach before the benefits of circular organization can be appreciated. This is not to say that feedback is impossible for dynamic organizations in the CORAL model; usually it is hard to avoid. However, as is demonstrated in Chapter 4, even without "closing the loop" and considering dynamic organizations one can build some highly interesting types of dynamic structures.

2.4.1 Meta-unit control

The above discussion was meant to motivate the general approach toward building artificial assemblers taken in this thesis. Small reactive units, in combination, can be assembled into complex, controllable devices. Ideally, one would be able to build and manipulate huge structures from very small, simple parts. As the size of structures grow, however, the complexity of specifying and controlling these structures also grows, generally geometrically.

One problem with this complexity is that it can overwhelm any finite pre-programmed unit, requiring the external signal approach also discussed in the above section. In practice, macroscopic units built with electronics can have huge information capacity, such that the limit above is not a practical concern, but there are efficiency and scaling problems. Every assembling unit is required to contain large amounts of memory and supporting hardware and software, which adds to the cost and power consumption of units. As units get smaller and capable of forming more intricate structures, the memory, hardware, and power penalties go up while the capacity to hold them goes down. For meso- and microscopic electronic parts, it becomes much harder to specify large assembled structures on the devices themselves. Chemical or nanoscale mechanical devices are currently even simpler, and while DNA or other chemical storage can achieve impressive data density, it is difficult to reprogram units of this size or use DNA for device-internal computation. Pre-programming dynamic units does not scale.

Once a device is assembled, it must be controlled (or control itself) to be useful, and the same calculation comes into play. As an assembled device grows, the complexity of controlling each unit individually grows linearly (or more), and this will overwhelm finite pre-programmed units in the same manner as above. While external signals (even broadcast-only) can theoretically encode any manner of control, ideally control would be efficient and involve as few signals as possible while allowing both large and small devices to be built. From this goal comes the idea of *meta-unit control*.

Meta-unit control is not a new idea in modular robotics, though it tends to be of more limited use in pre-programmed devices. In work with cubical unit-compressible modules, Rus and

Vona introduced the concept of *grains* - larger, multi-unit cubes with the ability to reconfigure themselves arbitrarily (Rus & Vona, 1999). Grains share the same unit-compressible operations as individual modules, and are able to reconfigure themselves arbitrarily using grain operations defined for grains of any size, though control of these modules is not discussed in a distributed way. The Molecule modular robotic platform has a similar concept called *tiles* (Kotay & Rus, 2000), as does the Shady robot using *metamodules* (Detweiler et al., 2007). Detweiler *et al.* propose a hierarchy of metamodules, where each level has a new control scheme. All three of these systems can create larger assembled units with simplified control properties, referred to here as *meta-units*. When an assembling system must create unknown devices, such units aggregate the control of many units into a single assembling shape.

These grain, tile, and metamodule meta-units are abstractions placed into the external controller; once the meta-unit structure is created, the same complex individual unit commands must be sent to move it, though now these signals are aggregated conceptually for the user. A major goal of the research in this thesis is to show that with careful choice of unit design and meta-unit structure, meta-units can *implicitly* realize meta-control through their aggregate structure *alone*. Instead of putting functional wrappers around sets of signals which happen to be useful controlling groups of units, one can design units which when assembled respond to the *same signals* as the individual units themselves. Both structure and function become self-similar, perhaps with a scaling factor. If designed even more carefully, meta-units can build meta-meta-units with the same control function, *ad infinitum* - an assembling *quine* (Hofstadter, 1979).

This results in a radical but intuitive compression of the control space, where larger instances of the same devices are controlled using the same signals. In addition, it is easy to give large structures simple behavior, a useful trick for the more structural aspects of a device. A similar idea exists in nanotechnology, though not explicitly for control, in *convergent assembly* (Merkle, 1997; Freitas & Merkle, 2004). Assuming processes exist which align a limited number of units together at any scale (perhaps a shaped lattice, chemical "tweezers," or robot arms), it is theoretically possible to grow geometrically larger and larger modules by feeding in small parts to a regular assembling mesh, assembling in parallel, and feeding these to the next higher level. As will be shown in Chapter 4, if the units are chosen correctly, no feed or mesh is needed, and any computational device can be created (Chapter 5). Instead of reaching downward toward atoms with progressively smaller "hands" (Feynman, 1959), one can also reach *upward.*

2.4.2 Other scaling approaches

Other scalable assembly methods have been proposed for robotic or swarm structures which do not directly use the concept of meta-units. One approach is to gradually increase the resolution of the desired structure as new modules are added to the system (Støy & Nagpal, 2004), implicitly generating a rough hierarchy from any shape. Other similar growth methods for biologically inspired agents (Nagpal et al., 2003), including morphogen gradients (Mamei et al., 2004), and robotic swarms (Cheng et al., 2005; Studer & Harvey, 2007) have also been proposed, though these approaches also lack means of control and require units with large memory. A new stochastic assembling approach to multi-level assembly is presented in (Mermoud et al., 2009), which uses heuristics to build multi-level approximations (much like (Thorsley & Klavins, 2008)). The

CORAL model does not require approximations to model structures, due to the nature of the components, but the definition and simulation of dynamically stable unit organizations may require similar model reductions.

The control of many assembling units through a high-level language compiled to distributed instructions has also been demonstrated in many forms, as was mentioned above. The basis of these languages include graph grammars (Klavins et al., 2006b), amorphous computing unit languages (Kondacs, 2003; Beal, 2005), geometric and stigmergic constraints (Grushin & Reggia, 2008), Petri nets (Costelha & Lima, 2008), or fact/action systems similar to classifier systems (Ashley-Rollman et al., 2009). In each case, the program in the high-level language is automatically broken into smaller pieces which are distributed to each unit. Depending on the implementation, the distributed rules may be unique to each unit type. Assuming types are finite, rules-per-type can be seen as equivalent to all units getting a the full set of rules predicated on state.

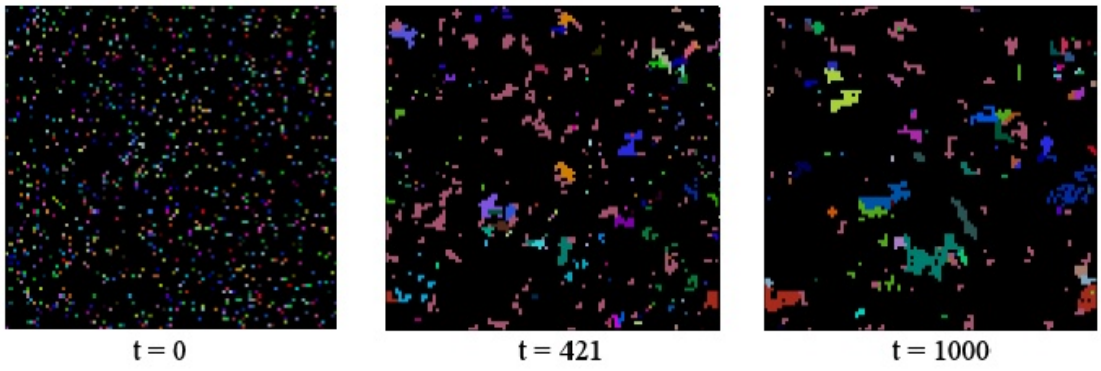
The idea is in many ways the inverse of meta-units and the assembly mechanism described in this paper, which describes mostly reactive distributed units building stateful, controllable devices. The difference is not simply a shift in perspective, however. The units postulated here are programmable *only through composition*, and cannot change their behavior directly if different devices are needed. This is a trade-off: one must build structures (potentially wastefully) to change the functionality of the atomic components. For example, using NOR units from Chapter 4 one must build a tree to implement the AND function, it does not exist atomically. On the other hand, the units themselves may be simpler and more uniform. It is conceivable (and rather interesting) to imagine a merge of these approaches in the future, where higher-level descriptions are compiled down to distributed operations, which are then themselves compiled down to particular structures with those functions.

2.5 Evolved approaches to assembled hierarchy

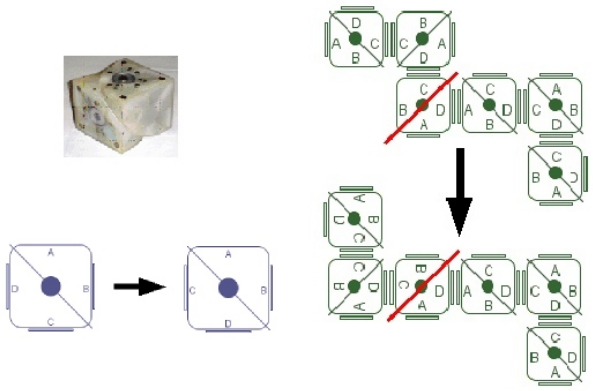
Evolution through natural selection, as the designer behind natural assembling systems throughout the biological world, can be considered the most successful generator of dynamical hierarchies known to date. In Chapter 6, evolution-inspired searches of CORAL model assemblers using virtual genetic algorithms are presented. There have been a limited number of previous studies using evolutionary simulations to generate systems with interactions at multiple dynamic levels, the earliest of which is the Tierra model (Ray, 1992, 1997) and the related Evita model (Bedau et al., 1997). The assembling robot platforms Molecubes (Zykov et al., 2007), ATRON (Østergaard & Lund, 2003), and Swarm-bots (Trianni et al., 2003, 2004) have also evolved control for multi-unit robotic structures using these methods. In general, however, the aim of these robotic studies is more to engineer flexible and fault-tolerant behaviors than to understand properties which lead to scalable construction.

2.5.1 Molecube replication

An exception is (Studer & Lipson, 2006), which demonstrates the emergence of replicating species of structures using abstracted Molecube behavior. The Molecube, as introduced above and pictured in Figure Figure 2.7b, is a cubical robot with electromagnets on each of the cube faces. An actuated diagonal cut through the cube allows the rotation of three faces of the Molecube with



(a)



(b)

Figure 2.7: Open-ended evolution of Molecubes (2.7b) in a cellular arena (2.7a) from (Studer & Lipson, 2006). The cellular grid is initally seeded with a small percentage of units with randomly generated controllers (arbitrarily assigned a color in (2.7a)). Each unit is capable of swiveling two halves with respect to one another and linking on four faces (2.7b). Individual units are incapable of motion, but by swiveling the units move connected units (up to a limit), and so structures may be mobile. Units can also overwrite nearby controllers with their own. Given a slow mutation rate of random overwrites of unit controllers with new random functions, the simulated Molecubes form continuously novel types of stable, self-replicating, and competing structures.

respect to the other three faces. In isolation the robots are basically immobile, however when assembled into structures they are able to dynamically reconfigure and even self-replicate (Zykov et al., 2007).

The Molecube simulation built to further explore this structural replication takes place in a mechanically realizable grid environment where robotic controllers for two-dimensional Molecubes are randomly generated and allowed to overwrite one another. To survive, units with controllers must form into structures which move and propagate themselves. Explicit tests were performed not only for individual replicators, of which there were many, but also for correlated species groups (though none were conclusively found). Structures of particular controller species often were found in a hierarchy of nested forms, where smaller, mobile forms later combine to form larger structures (Figure 2.8b).

The simulation uses a grid similar to those of other cellular automata models, though the grid in

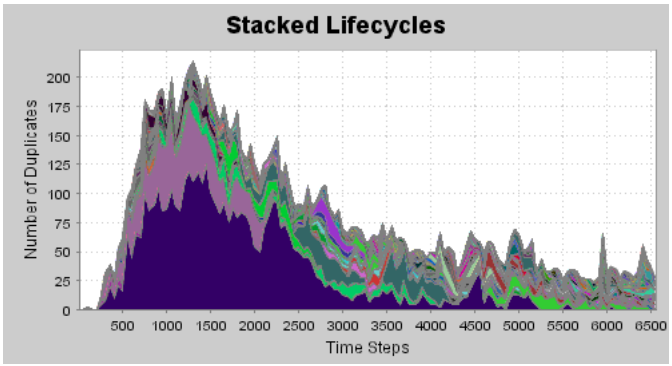
this case is populated with different molecube *controllers*, each contained in an implied molecube with orientation. There are generally many empty locations with no cube and no controller, loosely corresponding to the quiescent state of an ordinary CA. Controllers can be thought of as a mapping function from bits to bits, or equivalently as a binary classifier system. Random instances of this map are easily generated from a long binary string cut into input and output portions. Input to the controllers is simply the binary encoding of the von Neumann neighborhood of the molecubes (chosen because attachment between molecubes happens only on faces). The controller output in response to this input is also binary, and logically divided into three sections:

1. *magnet* - four bits indicating which faces (A, B, C, D) of the molecube should be able to link to other molecubes, i.e. whether the magnet is "on" or "off"
2. *swivel* - four bits indicating which half of the cube to swivel w.r.t. the other half. As can be seen in Figure 2.7b, swiveling is the way in which molecube structures reconfigure themselves. There are molecube types with different swivel cuts, grouping faces (AB)(CD) or (DA)(BC), and two bits control the output of each type of cube orientation.
3. *overwrite* - four bits indicating which neighbor controllers, in molecubes adjacent to faces (A, B, C, D), should be overwritten by a copy of the current molecube controller

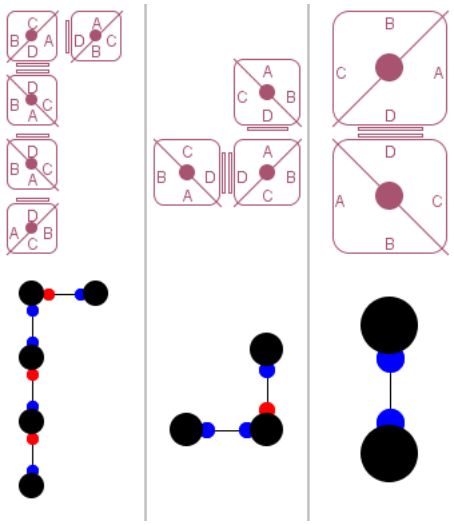
To visualize the environment, each molecube controller type is consistently assigned a random color in a 2D grid, with black as the quiescent state (Figure 2.7a). The simulation progresses by initializing random molecubes at random locations with a given density, choosing a random molecube order, and executing the molecube controllers in that order by providing neighbor input and processing the effects of the controller output. An example of such output might be to attach to a neighbor cube via a magnet bit, overwrite the cube's controller with an overwrite bit, and swivel two cube faces, moving all cubes attached to those faces (as in 2.7b). It is important to note that coordination between at least two molecubes is required for movement, a single cube attached to no neighbors cannot move.

These operations are designed to mimic the physically realized operations of real molecubes. As simulated time progresses, those controllers which can coordinate molecubes into forming movable structures are able to propagate themselves, and those which are not are often overwritten. To ensure diversity, mutation events happen spontaneously where a controller is completely overwritten by a random replacement - cube structures can protect themselves somewhat by continual overwrites but individuals are highly susceptible. This results in an implicit evolutionary drive toward fast, self-replicating structures in a physically realizable substrate, which was nearly always observed from initial states (see Figure 2.8).

Using this simulation, it was also found that replicating structures were often composed of one or more sub-structures, sometimes in near-fractal arrangement, which raised the question of whether multi-species structures could be observed as an even higher level of organization. Preliminary work toward this goal, which correlated over time the number of duplicated structures for each controller type (Studer & Lipson, 2006) indicated that there were in fact no (linear) correlations between duplicated structures of different species in the simulations tested, nor could any easily be generated by adding additional state and communication bits to the molecube controllers. This negative result had positive consequences, however, as it was the direct inspiration for the work undertaken in this thesis.



(a)



(b)

Figure 2.8: ”Stacked lifecycle” chart of a molecube simulation (2.8a), where the horizontal axis indicates simulation time steps (a time step is a full set of molecube executions) and the vertical axis indicates the number of duplicate structures of a particular prototype (2.8b) form containing a single type of controller. Growing numbers of duplicates indicate replication of cube *structures*, not simply propagation via overwrite of controllers. Color is the same as the controller color. As new species are created via mutation and propagate they are stacked on top of the older species, so the full area indicates the total number of duplicates in all species. In (2.8b), some duplicated species prototypes are shown with a colored graph representation - each face type is represented by a different colored node, which are implicitly attached to a central black molecube node and explicitly to other face nodes. These subgraphs can then be checked for isomorphism to detect duplicate structures. It is commonly the case that these duplicated structures are “sub-parts” of one another, as can be verified is the case above.

2.5.2 Open-ended evolution

Because there is no explicit evolutionary pressure toward an external goal, this type of simulation is classified as *open-ended* evolution. A major open problem in open-ended evolution, also mentioned in the open problems in artificial life (Bedau et al., 2000), is how complexity is generated. As described by (Ray, 1997), researchers seek to produce a "digital analog to the Cambrian explosion," in which multicellular organisms suddenly and inexplicably began appearing in the fossil record. The *in silico* equivalent has proven difficult to emulate fully, despite abstract, unconstrained environments (Sims, 1994; Ikegami & Hashimoto, 1997; Ofria & Adami, 1999; Lenski et al., 2003; Standish, 2003) and direct simulated attempts (Furusawa & Kaneko, 1998, 2002; Yoshida et al., 2005). A major problem is that there is not general agreement on how to compare complexity, particularly across models, or that complexity even increases (Bedau, 2009). New approaches to artificial evolutionary algorithms have been proposed as necessary, perhaps integrating recent biological discoveries in development and evolution (Banzhaf et al., 2006; Bedau, 2009) or extreme quantum parallelism (Standish, 2003). The exception to this, however, as mentioned above, is the emergence of self-replicating structures. These have appeared much more frequently than initially expected despite the lack of evolutionary pressure (Koza, 1992; Chou & Reggia, 1997; Studer & Lipson, 2006).

An approach partially avoiding these difficulties, using an earlier version of the CORAL model, was presented as (Studer & Harvey, 2008). While recognizing that increasing complexity is a difficult concept to define, most widely-accepted examples of complexification involve the composition of larger organizations from multiple smaller organizations. If the organizations at each scale are otherwise functionally identical, the evolved system, if not increasing in complexity, is at least at a threshold of dynamically renewing complexity by any measure chosen. Such systems are interesting not only theoretically, but also as a mechanism to achieve the above-stated goals of building scalable devices through assembly. The results from this work and extensions are further presented in Chapter 6.

2.6 A new model for assembly

The CORAL assembly model was developed to pull together the various strands of work presented in this chapter, taking the most compatible aspects from many of the above models and removing complications from particular domains. In particular, inspiration was taken from the FSM soup artificial chemistry approach, new rule-based biochemical models, and modular robotic interactions. Aspects of assembly models it was considered vital to support in the CORAL model are:

- Asynchronous, local interactions - assembling units are independent and act subject to local connectivity constraints
- Asynchronous, finite-state units - assembling units have highly limited memory
- External information source - reconfiguration information cannot be preprogrammed into units
- Conservation of matter - atomic assembling units must be reconfigured, not created or destroyed

- Inter-unit interactions identical to intra-unit interactions - assembled devices are not hybrid constructions but "first-class citizens," equivalent to larger atomic units

The first three of these constraints were discussed above in Tables 2.1, 2.2, and 2.3. The fourth, conservation of matter, is an extremely important consideration for real assembling systems, and to a large extent it seems the ease of creating hierarchical systems is related strongly to whether or not atomic units are infinitely malleable or may reproduce (such as in abstract artificial chemistries). Assembly results from the CORAL model cannot depend on a lower level of replication or construction, since the highly-related properties of assembly are the focus of the model itself. The fifth constraint is perhaps the most unconventional, but the idea is shared by the new rule-based biochemical and grammar-based robotic models. As has been shown by assembly research targeted at specific domains, the analysis of huge constructs becomes qualitatively different (and often intractable) if the interactions between assembled units are not captured as strictly as those in the units themselves. By making the inner and outer interactions equivalent, one can use a single set of mathematical tools to analyze, control, and compare behavior across scales. The next chapter describes the implementation of the CORAL model and the Petri net formalism which captures the above requirements.

Chapter 3

The CORAL Model

To scientifically investigate assembling systems, especially large systems with mixtures of composite and atomic pieces, one must first define in a rigorous way the units which are able to undergo assembly and the mechanisms by which these units interact. In this chapter, we define precisely the operation of an idealized environment and unit behavior which we declare is an instance of assembly: the CORAL model. Designed with simple but extendable assumptions compatible with realistic environments, the model combines the unique properties of controllable interactions via background signals and complete equivalence between external unit communication and internal unit processing. By blurring the line between composite and atomic units, the CORAL model allows the investigation of controllable assembly as a process without reference to any particular scale. Created for this thesis, an early version of the model was presented at the ALife XI conference (Studer & Harvey, 2008).

3.1 Overview

The CORAL (Computational Organization and Regulation over Assembly Levels) assembly model captures the notion of discrete, identical assembling components as they interact in a well-mixed environment. Intuitively, the simulated environment is a "sea" or "soup" of many identical parts with a limited number of assembly ports that may open or close. There is a shared background signal which is detected at each timestep by each unit, and this background signal may modify the interactions of all units in the simulation *simultaneously* (noise and asynchronicity exist, but are modeled in unit internals) (see Section 3.5.1). Through externally-directed changes of the background signal and designing simple units which respond to these changes by opening ports, units may be assembled into extremely large structures. To extend the oceanic analogy further: we can modify the salinity, but we cannot touch individual parts. There is no mechanism in the CORAL model by which an individual unit may be addressed separately from any other, units are *indistinguishable*.

As discussed at the end of Chapter 2, the main motivation behind the design of the CORAL framework is to see how much complexity one can *offload* to this broadcast background signal and *remove* from individual parts and the environment. In many real assembling systems such as

artificial DNA computing or robotic swarms, broadcast information is cheap to create and control compared to the redesign of individual parts. In others, such as developing biological organisms or chemical networks, non-local environmental manipulation may be the only external control possible. The explicit recognition of external influences is also required of systems which are meant to be dynamically controlled, and background signals are a simple model of these influences.

This design, which strongly limits the complexity of individual parts but allows complex design to be inserted via external signals, also avoids many of the philosophical issues raised when describing a system as generating structure at multiple scales (Rasmussen et al., 2001b; Groß & McMullin, 2001; Rasmussen et al., 2001a). For example, is a larger structure simply an aggregate of smaller units acting independently, or does it deserve description as a new entity? With background signals, structures at larger scales can be unambiguously and pragmatically identified: a set of devices with the same types of interactions in response to the same signals.

Units interact only when assembled with one another in the CORAL environment, which differs from traditional cellular automata (CA)-type models also used to model discrete, distributed systems. The CORAL assembly process can form dynamic networks (or structures) of any topology, restricted only by the connectivity of the atomic units. These assembled structures are designed to be directly comparable in complexity and operation to the atomic structures themselves. Large and complex connected assemblies can be reintroduced into the simulation as atomic units that contain many sensors and potentially higher connectivity. In a CA model, large regions of correlated cells (a structure, for the sake of this discussion) can have jagged and fluid external boundaries that are qualitatively different from a single cell neighborhood. If one of these regions was encapsulated as a single cell in a larger regular automata, the boundaries may have no direct equivalent. In addition, CA cells and structures either update synchronously, in which case CA structures can be a poor approximation of real assemblies made of many mostly-independent components, or asynchronously, in which case CA structures also behave internally in a qualitatively different way from the synchronously-updating cells of which they are composed.

The CORAL model trades the simplicity of a predefined topology and synchronized operation for simplicity in comparing units across scales. Other artificial chemistry models have relaxed one or both of these requirements, often in domain-specific ways. However, the combination of simple, generic environmental interactions and single mechanism of external control in the CORAL model serves to link as closely as possible the computation of simple but plausible assembling devices with the input driving these devices. Due to the use of Petri net event synchronization as a basis for intra-unit control *and* inter-unit interactions, this link holds as assemblies grow to arbitrary scales.

3.1.1 Model components

It is useful to point out again the overall components of the CORAL model, to serve as a useful reference point going forward. Further sections in this chapter provide more complete details and examples, but an introductory listing the components includes:

- a single, *labeled Petri net* which defines unit behavior. Petri and C/E nets are discussed first, in Section 3.2, but this section may be skipped if one is already familiar with the Petri net computational model.

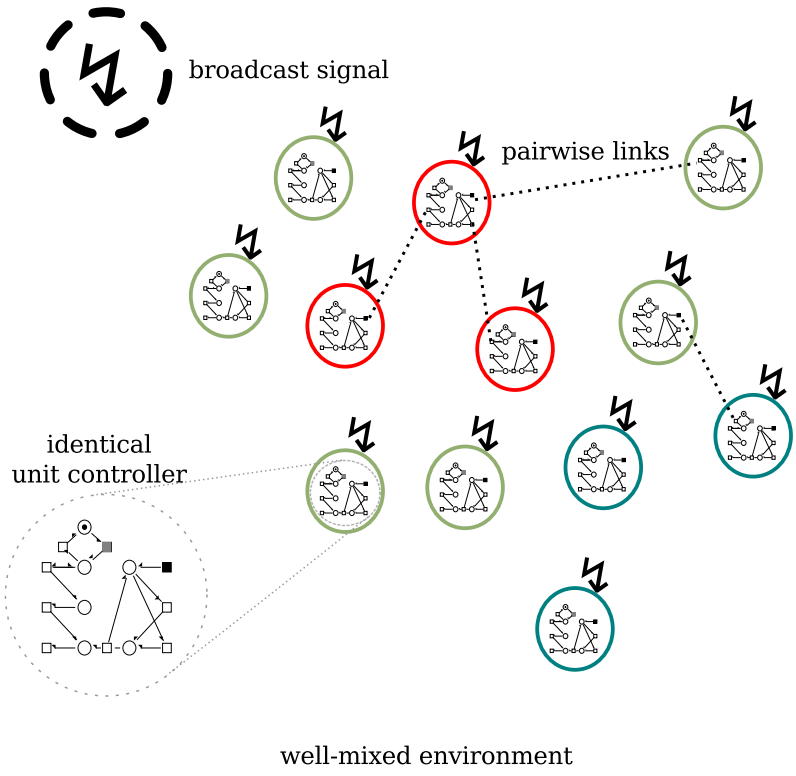


Figure 3.1: A diagram of the core features of the CORAL model. Well-mixed units "float" in the environment, each with identical controller but potentially unique state (indicated by color). Units may be linked to one another via pairwise links between ports which are enabled and disabled by the unit controller. A background, broadcast signal is always present in the simulation and affects each unit in the same state identically.

- a set of many identical, asynchronously executing *units* with state and complementary ports. Section 3.3 describes in detail how the behavioral C/E net, mentioned above, controls the interactions of these units in various states.
- a single well-mixed *environment* in which all units reside, which is responsible for the creation of links between units in particular states. Section 3.4 defines the CORAL environment.
- a single *broadcast signal*, consisting of a single symbol at any given time, which may enable particular state transitions defined in the behavioral net. This is again a novel feature of the CORAL model, and addressed in Section 3.5.

In a compact description, the CORAL model is an *asynchronous* model of many *anonymous*, *identical*, *interacting* computational units. Figure 3.1 highlights the main components graphically, and there are further examples throughout the chapter drawn in a similar way. One can match the features with the list above: the units are all identical (each contains the same Petri net, though may have different markings), current environmental links are drawn as dotted lines, and the current symbol of the broadcast signal is placed in the background and circled with a dotted line. The position of units when visualized does not affect behavior; only the topology of connections is important.

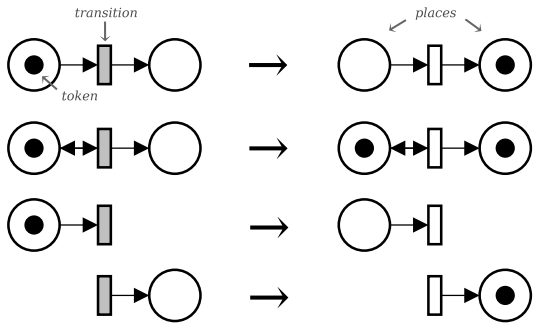


Figure 3.2: Summary of basic Petri and C/E net operations. Rectangles indicate transitions, white circles are places, black circles in places are tokens. When treated mathematically, Petri nets are often formulated such that transitions with no-input, no-output, or input-and-output places are ignored (since these cases can be emulated by other constructions), but the nets can be more naturally drawn and understood graphically if these types of transitions are allowed.

3.2 Petri nets and C/E nets

The basis of the CORAL model is derived from a Petri net variant - the condition-event (C/E) net. Petri nets, as a formal, well-understood model of distributed computation with graphical format, have recently become an interesting platform from which to investigate robotic and biochemical systems. Though much work focuses on Petri nets as an intuitive model for multi-step planning and manufacturing tasks (DiCesare, 1993), other research has expanded their application to the coordination of robot teams (Wang & Saridis, 1993; Sheng & Yang, 2005; Costelha & Lima, 2008), complex biochemical reactions (Peleg et al., 2005), and even as a computational basis for the relativistic universe (Zuse, 1969; Petri, 1996, 2008). The primary advantage of Petri net systems in these areas is the ease of modeling concurrent (i.e. independent) and synchronized operations. Highly interleaved processes can be represented as a network of tokens on a simple labeled graph. Assuming the graph is a compact representation of the configuration space of some system, for example chemical species or robot states, the resulting Petri net encompasses all the behavior in the system in an intuitive, yet computationally rigorous way.

The description of Petri nets below is derived largely from (Peterson, 1981), (Jensen & Rozenberg, 1991), (Reisig, 1992), and (Petri, 1996). Formally, a Petri net can be described as a *directed bipartite graph* of *places* and *transitions,* along with a distributed *marking* (Figure 3.3). Places hold the state of the modeled system as *tokens*, while transitions determine how tokens move between places. Each place may contain one or more tokens, and a full listing of the number of tokens in each place is again the *marking* or a *constellation*. If all the places attached to the incoming edges of a transition have tokens (or there are no incoming edges), the transition is *enabled* and may *fire* (see Figure 3.2). When a transition fires, a token from each incoming place is removed and a token is added to each outgoing place. This changes the token marking, which might then enable other transitions. It is possible, and normal, for multiple transitions to be active at the same time; the choice of which transition to fire is nondeterministic. Figures 3.4 and 3.5 illustrate a limited type of Petri net, a C/E net, in operation.

One can think of a Petri net as a kind of token game, where tokens move around the graph via transitions into new configurations (and may be created and destroyed). As the game progresses,

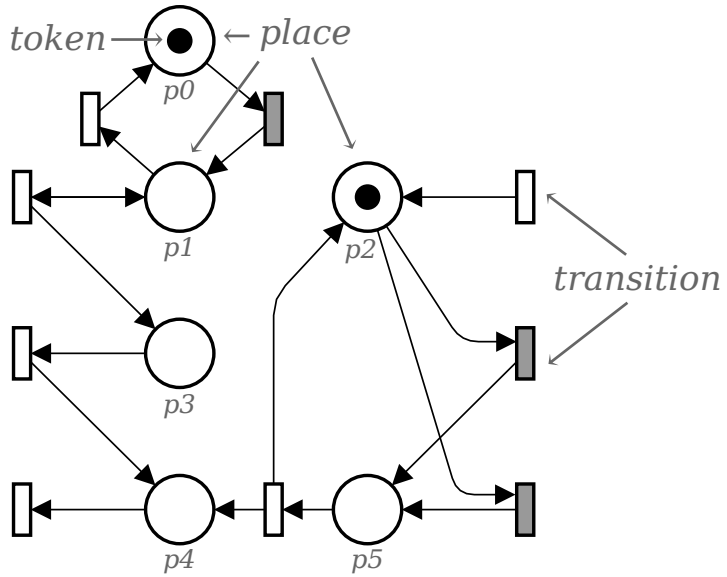
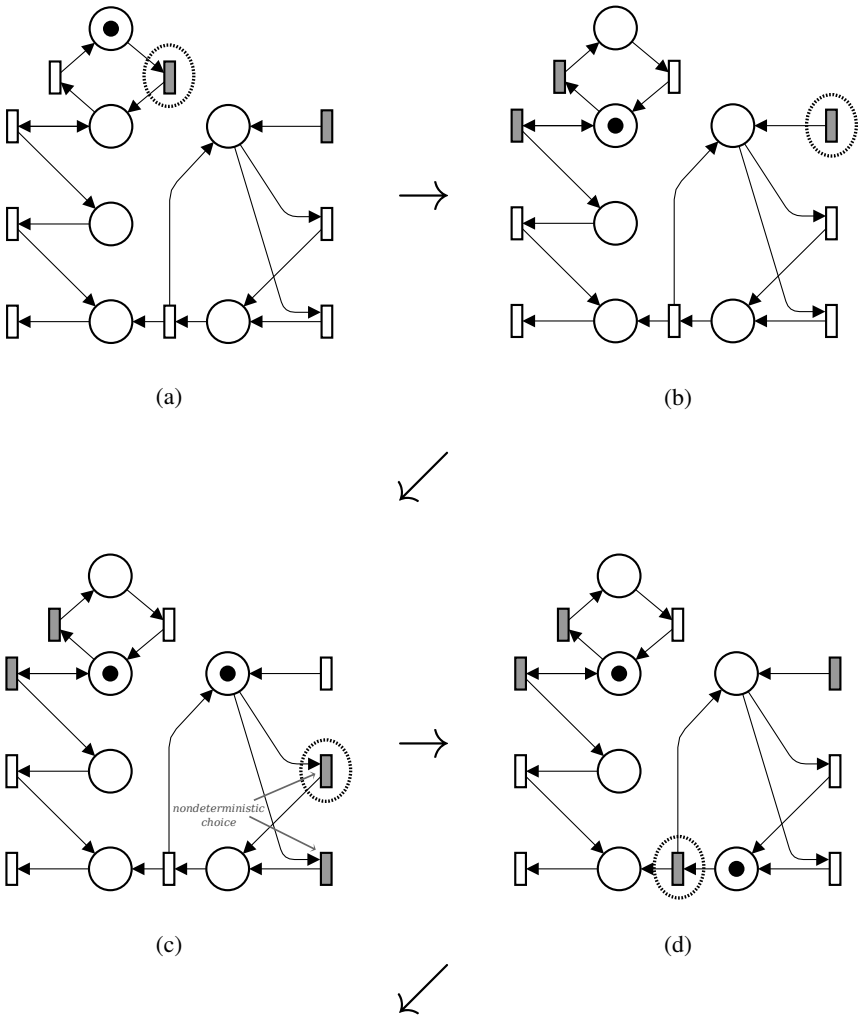


Figure 3.3: An example Petri or C/E net diagram. By convention, Petri net places are depicted as circles and transitions as squares or rectangles. C/E nets are limited to a single token per place, while general Petri nets may have unlimited numbers of tokens per place. Arrows are directed edges indicating the incoming and outgoing places and transitions for each node (double arrows indicate both incoming and outgoing edges). The graph is bipartite, therefore transitions may only have directed edges to and from places and places may only have edges to and from transitions. The marking of the C/E net above is the set of all places containing tokens, i.e. $\{p0, p2\}$. The convention adopted in this thesis is for active transitions to be colored gray.



(continued in Figure 3.5)

Figure 3.4: An example C/E net execution sequence of the net depicted in Figure 3.3. Transitions which are enabled and may fire are colored grey, those which are chosen to fire in this sample execution are circled with a thinly-dashed line. In the initial marking, shown as (3.4a), only the topmost place contains a token. Two transitions in (3.4a) are enabled to fire, however; the central enabled transition because the token is in the incoming place and the rightmost enabled transition because it has only an outgoing edge pointing to an empty place with no tokens. In (3.4b) the central transition has fired, placing the token downward one place node and enabling two other transitions. The choice of which transition to fire is made nondeterministically, and (3.4c) and (3.4d) depict the C/E net marking assuming the rightmost and right-bottom enabled transitions have fired, respectively. The sequence continues in Figure 3.5.

(continued from Figure 3.4)

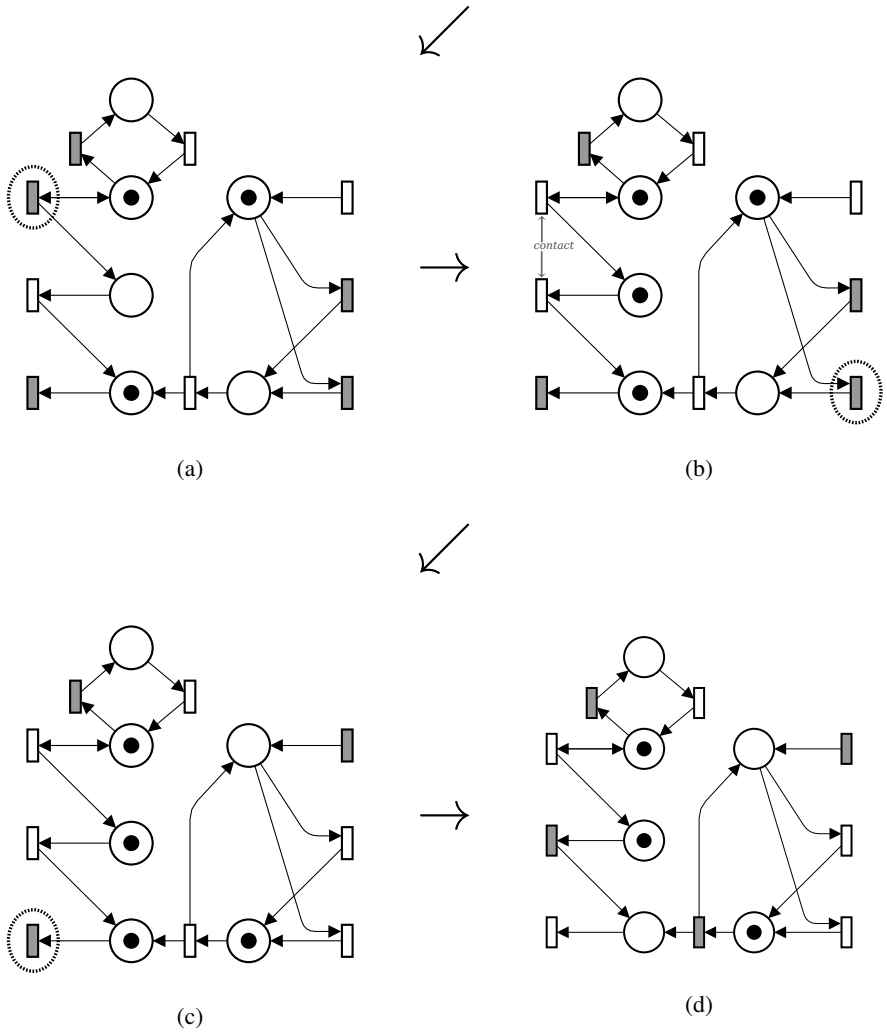


Figure 3.5: Continued execution of the C/E net depicted in Figure 3.4. The marking in (3.5a) was the result of the central-bottom enabled transition firing from (3.4d) in the previous figure. The leftmost, bottommost, and bottommost enabled transitions have fired to create the markings in (3.5b), (3.5c), and (3.5d), respectively. In (3.5b), the enabled transition with both incoming and outgoing edges from (3.5a) has fired, leaving the double-edged place unchanged but also adding a token to the place below. This same transition and the one below are not enabled in (3.5b) and (3.5c) because of *contact* - firing would result in more than one token per place, so is disallowed under the rules of C/E nets. In 3.5d the lower-left token has been removed, and so the middle-left transition can become enabled.

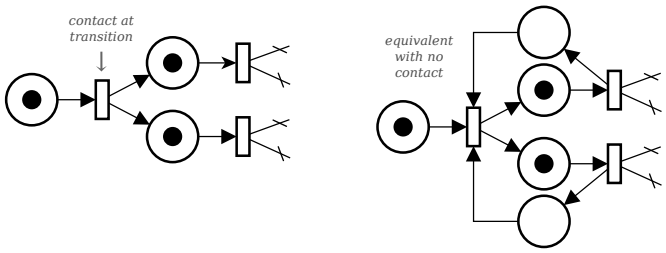


Figure 3.6: Transition with contact in a C/E net. Because the outgoing places are full the transition cannot fire. Such a net can always be emulated by an equivalent construction where no contact is possible. Potential contact places are duplicated to create complementary places with opposite edges to each transition. In this way, conditions on the outgoing places can be replaced with extra conditions on the incoming places.

certain places may accumulate tokens or be emptied of them, and the net may or may not deadlock. The network can be studied for certain conditions or "fact transitions," represented as particular transition firings, and can also be analyzed for *boundedness*. Boundedness refers to the theoretical maximum number of tokens which may appear in any place, so that a k-bounded net will only ever have k tokens at once in a place. Petri nets which are 1-bounded are also called *safe*.

There are many different varieties and extensions of Petri nets, but in this thesis only *safe* and *ordinary* Petri nets are used - those with unweighted (or weight=1) edges and restricted to a single token per place. (Loops are possible, however, where a place has a directed edge to a transition, and that transition has a directed edge back to the same place.) These restricted nets can also be described as condition-event (C/E) nets (Reisig, 1992; Petri, 1996) or predicate/transition (PrT) nets (Jensen & Rozenberg, 1991) to emphasize the close connection to formal logics. The C/E net convention is used here.

The single-token distinction is an important consideration for the modeling of realistic assembling units. Arbitrary Petri nets, as we have defined them above, may assume an infinite number of states, since each place may contain any number of tokens. Real devices, however, do not necessarily have an infinite number of distinguishable configurations that can be used for computation, particularly if the devices are small. Ideally our model should reflect this finite state limitation. The modification to the Petri net definition above, adding a *capacity* to each place, realizes this restriction and defines a C/E net. Any finite capacity could be used, but for simplicity all places have a maximum capacity of one token. With a one-token capacity, C/E net markings can be represented as binary strings, and these are easy to analyze and quickly simulate. C/E nets are Petri nets, but certain unbounded Petri nets cannot be implemented as finite C/E nets.

It is important to clarify exactly how this capacity restriction affects C/E net execution. C/E nets may have transitions that are blocked by full places, called *contact*, which is not an issue in general Petri nets. The C/E rule for transition enablement is that a transition may fire if and only if each place attached to an incoming edge of the transition contains a token, and each place attached solely to an outgoing edge of the transition does not contain a token. In other words, a transition firing can be blocked by outgoing place tokens, not including looped places (see the C/E net execution from Figure 3.5b and Figure 3.6). After a transition fires, each place attached solely to an incoming edge has no tokens, while each place attached to an outgoing edge contains

a token. Again, as might be intuitively expected, tokens move from source to destination places except for looped places which instantaneously give a token and then receive it back. For a modern mathematical treatment of both Petri and C/E nets, the reader is referred to (Reisig, 1992; Petri, 1996). It is important to note again that there is a simple transformation of C/E nets to safe nets without token restrictions (by emulating each place with potential contact as two unbounded complementary places (Peterson, 1981)) (see Figure 3.6), so the tools of Petri net analysis are always available for C/E nets.

3.2.1 Comparison with other approaches

Cellular automata for many years have been a traditional choice when modeling highly distributed systems, so the use of a novel framework benefits from a comparison. In part, our study of assembling systems derives from artificial chemistries, and many artificial chemistries have abandoned a rigid grid topology in favor of either well-mixed or physical interaction environment. In the case of the CORAL assembly model, well-mixed interactions are used because they are physically plausible, simple to simulate, and do not depend on the topology of the generated structures. Other related work in formal artificial chemistries tends to use this approach (Fontana & Buss, 1996; Dittrich & Banzhaf, 1998; Crutchfield & Görnerup, 2006; Salzberg, 2006), as do some formal models of chemical processes (Winfree, 1996; Danos & Laneve, 2004; Dittrich & di Fenizio, 2007).

Cellular automata use finite state machines (FSMs) as the basic unit of computation, as do many other assembling systems. FSMs and the closely related finite state transducers (FSTs) are well-understood computational structures (Kozen, 1997), but ignore two critical components which makes their use difficult in this context: parallelism and synchronization. A reactive unit modeled as a FST is always in a particular state at any given time, and at some future moment produces and/or consumes a signal to move to a next state. If two FST units are linked and communicating with one another, one must produce a signal while the other simultaneously consumes the signal (otherwise extra state is required and the unit is not modeled entirely as an FST). The two FSTs have synchronous state transitions for this event (see Figure 3.7). While it is entirely possible to define a model of this type, such a device can be modeled equivalently as a Petri or C/E net (Peterson, 1981).

Petri nets generalize sets of state machines to include synchronous and asynchronous transitions. Two FSTs acting independently are equivalent to a Petri net with two disconnected and singly-marked components, while two FSTs sharing a synchronous transition are equivalent to a Petri net in which singly-marked components share a Petri-transition. Two Petri net components placed side-by-side are trivially a larger Petri net, which makes it simple to analyze aggregations of multiple nets. This compositional property is especially useful when comparing the complexity and behavior of assembled units over multiple scales. The AlChemy model (Fontana & Buss, 1996), which uses λ-calculus expressions as primitives, as well as the "process soup" models (Salzberg, 2007; Görnerup & Crutchfield, 2008), which use FSTs, share this property to a degree in that derived structures are modeled identically to the initial components. The derived λ-calculus programs or FST structures are composed using highly nonphysical information processes, however, not through simple addition. Certain rule-based chemical models (Danos & Laneve, 2004; Faeder et al., 2005) and graph grammars (Klavins et al., 2006b) also support deriving composite

structural rules from individual rules, though again the functional composition process is not as straightforward.

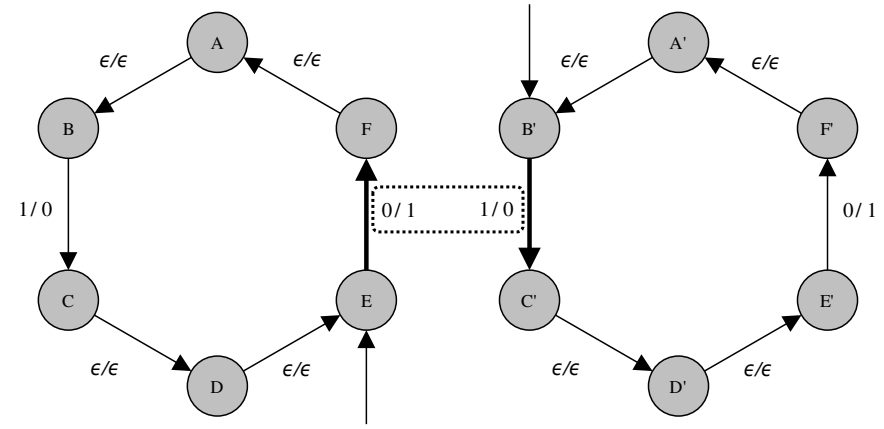
Equivalent formulations

C/E nets may also be represented equivalently using classifier systems, neural networks, and graph rewriting formalisms. If one considers the vector of places in a C/E net as a string of bits, C/E transitions can be represented as a function mapping a set of bit strings (with wildcards) to a different set of bit strings. In fact, the implementation of C/E nets written for this thesis uses the binary representation for speed. This is also known as a Holland classifier system (Holland, 1992), of which C/E nets are (at least) a subset. The construction is similar to that of (Reid, 1998), in which each transition is mapped to an input and output vector action rule. Each position in the input and output vector corresponds to a place, and the number in that position determines the input tokens consumed and output tokens produced. Given single-token capacity limitations, the vectors above collapse to binary strings. Classifier systems themselves can also be represented using a subclass of feed-forward artificial neural networks (ANNs) (Smith & Cribbs, 1994), which may seem unsurprising in this context given the intuitive similarity between Petri nets and recurrent ANNs. Incidentally, the artificial chemistry of McCaskill also uses a classifier pattern matching as a condition for reactions (McCaskill, 1988; Dittrich et al., 2001), though the results of interactions can change these classifier rules. As the abstract bit-string rewriting system, or more directly as a dynamic graph, C/E and Petri nets can also be easily encoded into rules of graph grammars (Corradini, 1995).

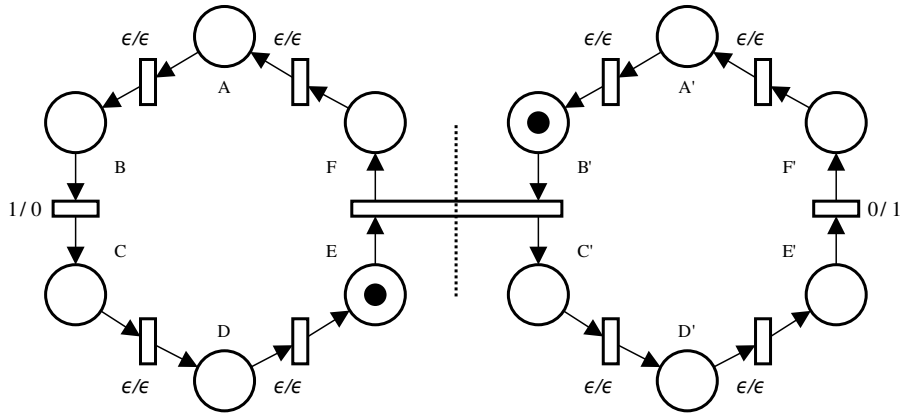
As a result of this mathematical closeness, C/E net models of assembling units introduced in this thesis are amenable to being described in a variety of different, but related, formalisms. This flexibility is useful, since assembly research has been previously studied using many different models. Also, in analysis, often it is useful to visualize a C/E unit as a simple state machine, ignoring concurrency, or *unfolding* (McMillan, 1995; Esparza et al., 2002), emphasizing concurrent pathways. At other times the full Petri or C/E representation more clearly shows the shared interactions between structures. From a philosophical perspective, often it is not clear in assembling systems what the different "parts" are once a variety of structures have formed, making it tempting but misleading to analyze signals sent between easily identified components. By defining communication more broadly as a synchronized relationship between two actions, the idea can encompass not only sending signals (linking send-token and receive-token actions/transitions) but also exchanges of information not easily identified with a "sender" or "receiver". To use an analogy, commuters living in different suburbs of London and leaving at different times can often see familiar people commuting on the trains, simply because everyone is influenced by the discrete train schedules. No secret messages are involved (though it may seem so at times), instead this is a property of the train system itself. The Petri net notion of synchronization captures this idea concisely, and from a finite automata perspective.

3.2.2 Related Petri net assembly models

Other assembly research has used Petri nets to control or model assembling systems. Extending the classical uses of Petri nets to model manufacturing processes, it has been shown that one can create a top-down dynamic program for robot groups (Milutinovic & Lima, 2002) and break the



(a)



(b)

Figure 3.7: Two synchronized FSTs and the equivalent Petri and C/E net construction. In (3.7a), two isomorphic FSTs with input/output signals on each transition arrow are in states E and B' (indicated by an arrow without a source) with complementary state transitions. At some future time, both FSTs may simultaneously transition to states F and C' respectively, consuming and producing 0 and 1 signals from one another (indicated by the dotted box). Pictured in (3.7b) is the equivalent structure as a Petri net. Directed edges from places E and B' indicate that tokens at these places are necessary for the shared central transition, which after firing places tokens at the F and C' places. The Petri transitions in (3.7b) have been labeled with the input/output signals from the corresponding FST transitions of (3.7a), including transitions with empty signals ε. The central transition has no label since the shared signals are produced and consumed internal to the FST pair. If this transition is broken into two independent Petri transitions (indicated by the dotted line), the result is two identical Petri nets which emulate the original FSTs without synchronization.

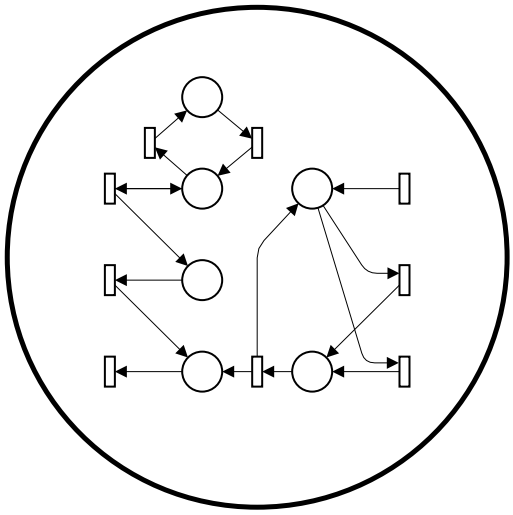


Figure 3.8: A sample atomic unit, visualized as the C/E controller inside the unit circle boundary. The controller is the C/E net from Figures 3.3, 3.4, and 3.5.

program down automatically into robot actions (Costelha & Lima, 2008; Palamara et al., 2009). The opposite operation, reconstructing of the behavior of multiple units as a behavioral Petri net, is also possible. Klavins applies this operation to assembling units (Klavins, 2006), and there are many examples of reconstructing Petri net models of biochemical systems (Peterson, 1981; Peleg et al., 2005).

In the work mentioned above, however, the actions and transformations of entire systems are modeled as Petri nets. The individual units may, and generally do, interact using some other system entirely - compiled C++, graph grammars, or chemical interactions. Distributed processing at a unit level is not considered, and the work in this thesis instead emphasizes these distributed units. Petri nets are used in the CORAL model to represent generic, discrete computing ability internal to the units, *not* sequences of external actions or group properties (though this could be done as well). Petri net synchronization is also used as the core communication ability in the CORAL model, much the way named signals are used as the core of the π-calculus. This approach applied to assembly is novel to the knowledge of the author, and provides an elegant and physically plausible method of generating nontrivial structure in a formal way.

3.3 Atomic Units

Now that the core C/E model has been defined and motivated, the particulars of the interactions and units in the CORAL model can be described. As introduced above, a CORAL simulation consists of a large number of atomic units interacting in the well-mixed "sea" or "soup." Each unit has an identical onboard *controller*, modeled as the simple Petri net variant described above: a C/E net. Like all Petri nets, the atomic units' C/E nets operate by moving tokens in places via transitions to other places. All of a unit's C/E net places and most of the transitions are internal to the unit itself, and unless affected by signals or assembly these act exactly as they would in any other C/E net. A unit's state is simply the current marking of these places. A sample atomic unit containing the same C/E net used in the previous examples above is shown as Figure 3.8.

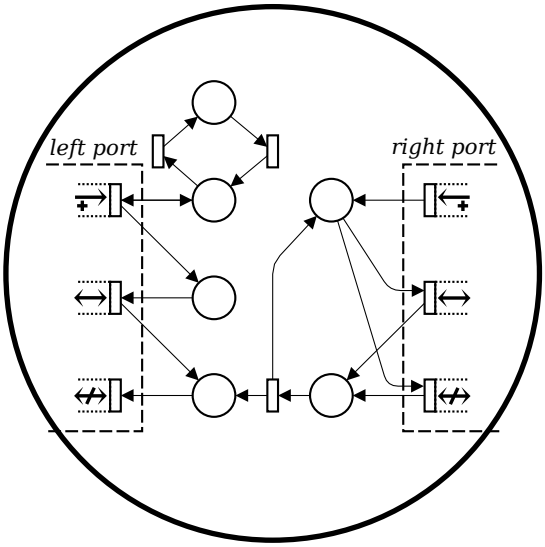


Figure 3.9: Example assembling unit from Figure 3.8 with *linking*, *synching*, and *unlinking* transitions assigned and labeled in two complementary *ports*. Linking transitions are indicated by a open dotted box containing an arrow with addition symbol, synching transitions by an open box containing a two-way arrow, and unlinking transitions indicated by an open box containing a two-way arrow with strikethrough. The open dashed boxes enclosing left and right transitions group these transitions into two complementary ports (multiple ports and port types can be defined, but only a single port pair is shown here for simplicity). Since there is only a single pair of complementary left/right ports the labels are omitted in later figures, though in the more complex examples in later chapters both ports and transitions are often referred to by labels. Note that equal numbers of each type of transition are assigned to each port. With basic assembly operations assigned to these transitions, the unit may now attach, communicate, and detach from other units in a CORAL environment, which will be simulated later in the chapter.

3.4 The Environment

In order to eliminate the possibility that the simplicity of the atomic units is simply due to the hidden complexity in particular environmental interactions, the CORAL model takes a minimal approach to representing the environment. As in other artificial chemistry models, assembling units are well-mixed, with no spatial position or orientation. Assembly takes place through complementary *ports* on individual units, and open ports in the environment are simply paired together at a constant rate. Atomic units can only specify which ports are open, but they are in no way able to directly influence the complementary units to which they will be joined. Imagining each unit as a protein, the units may catalyze or join when randomly encountering another unit but cannot choose which units they encounter. This behavior has the advantage of simple analysis and simulation, though real assembling systems appear in diverse environments and no single formalism can capture all possible types of environmental interactions. By simplifying these interactions to the greatest possible extent, the CORAL model can better clarify the minimal discrete logic required for flexible assembling units responding to only a single source of external information.

Multiple types of complementary ports can be defined in a given environment instance, and

each port pair may be assigned labels ($In \rightarrow Out$, $Left \rightarrow Right$, $Top \rightarrow Bottom$, etc.). These port types approximate the different interaction types inherent to a particular kind of assembling device such as a robot or chemical. A unit in the environment may then have zero or more *instances* of each port *type*, each port instance assigned to a unique set of unit controller transitions. Port instances correspond to particular interactions supported by a single device. Whether a unit port is open or closed depends on the activation of assigned transitions inside the unit, which in turn depends on the unit controller marking. A mapping is defined *a priori* assigning particular transition sets to particular ports, where certain transitions will, when able to fire for a certain interval of time, enable the external port. These are called *linking transitions*, and their activation essentially corresponds to a hypothetical device being in a receptive state for assembly or communication. There may be many of these linking transitions which enable the same port type (but different ports) on a particular CORAL unit. The idea is probably best captured visually, and Figure 3.9 shows an example assembling unit in which two complementary ports have been defined and the transitions assigned to each. Each complementary port must be assigned the same number and types of transitions, otherwise pairing the transitions is not well-defined.

If there are two open ports of complementary type in the environment due to linking transitions being enabled and *holding* (described below), a *connection* or link may be created between the two units. When created, the unit markings are changed by the simultaneous firing of the linking transitions which activated the connected ports, as shown in Figures 3.10 and 3.11. The new environmental link *synchronizes* pairs of transitions between the units, only allowing the *synchronized transitions* to fire if both transitions can fire simultaneously. Synchronized transitions behave exactly as if they were a single transition in a larger, composite C/E net (since synchronization is a primitive ability of transitions), making the linked unit effectively a larger atomic part.

Linking Petri nets via transitions is described in (Peterson, 1981) as a language-exploring construction and reviewed in (Reisig, 2009) across various domains, but the idea has not yet been applied to dynamic assembling devices. Intuitively, however, there are advantages to thinking about assembly this way. Challenging philosophical and mathematical problems arise when trying to decide how to define separate "objects" in a system with dynamic structure. Often new assumptions or unit limitations must be added to identify larger structures, even when they seem natural to a particular system (Ray, 1992; Rasmussen et al., 2001b; Lenski et al., 2003; Ewaschuk & Turney, 2006; Hutton, 2007), or conversely larger structures may be the result of many irreversible procedures or a process of computational development and no longer composed of atomic individuals (McCaskill, 1988; Fontana & Buss, 1996; Ikegami, 1999; Salzberg, 2007; Görnerup & Crutchfield, 2008). The linked Petri net approach sidesteps these issues in an elegant way, effectively declaring that two or more connected units are both a collection synchronized individual entities *and* a well-defined and identically modeled whole. Each structure can trivially be reduced to component parts, but this operation does not impact the status of the structure's unity, which can be decided (or not) by any applicable method. Also note that the slightly strange phrase "synchronized transition(s)" is sometimes used for paired transitions to emphasize the parts/whole duality.

In the above discussion, linking and synchronized (*synching*) transitions were defined for assembling units, but composite units must also sometimes break. As opposed to linking, unlinking

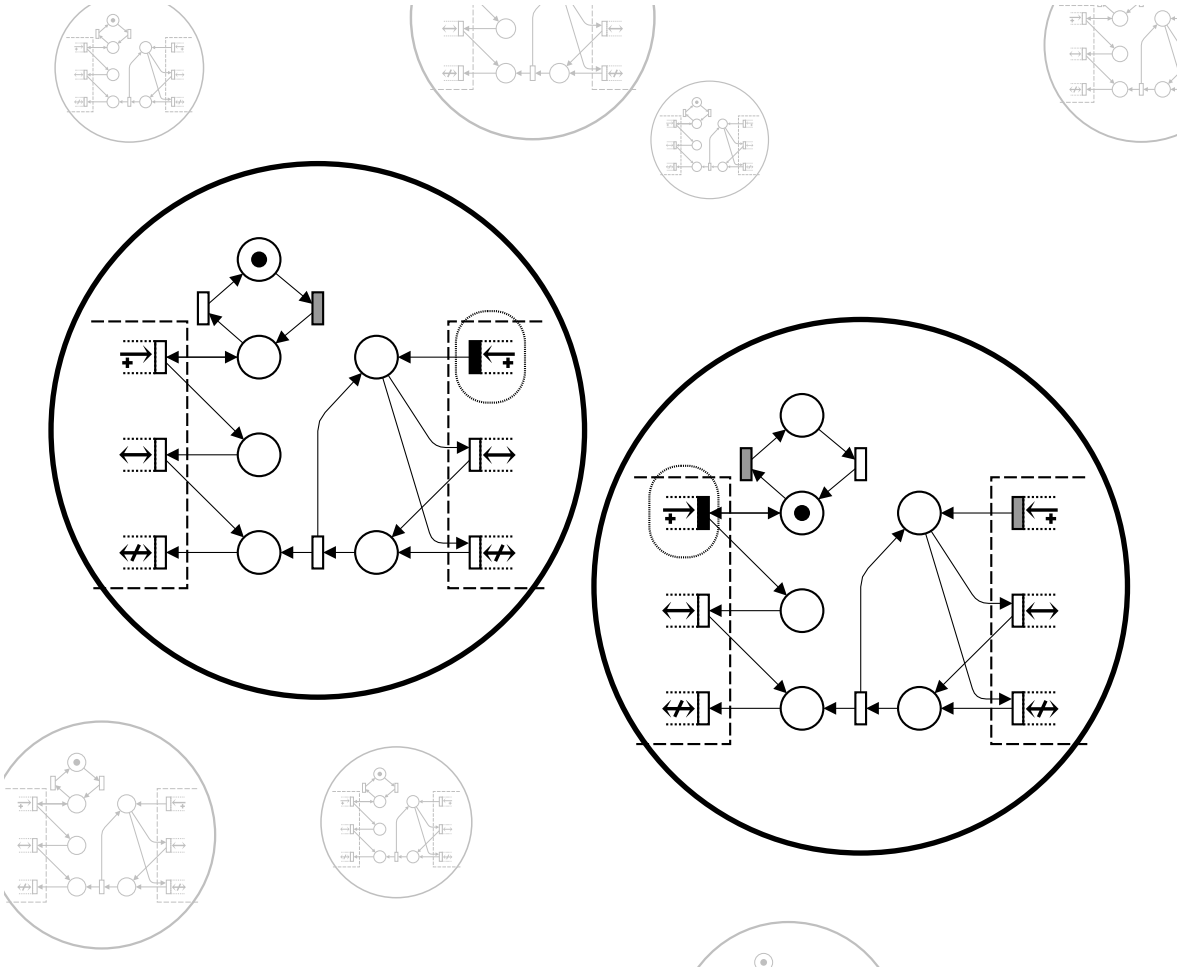


Figure 3.10: Several example assembling units from Figure 3.9 in the topological CORAL environment. The largest units have complementary link transitions enabled and are *holding* (colored fully black) because the environment has not yet allowed assembly. At some future time, these units with holding link transitions will have complementary ports paired and assembled together, allowing the currently holding transitions (circled in a thinly dashed line) to fire as merged transition(s) (shown in Figure 3.11). Other link transitions may be enabled but not yet holding because they have not waited long enough (discussed in Section 3.6).

is controlled by the connected units themselves. When both units have appropriate markings, particular synchronized *unlink* transition(s) may fire, *not* directed by the environment. This again models the same procedure in real devices, where robots or proteins must be in particular states to disengage from one another. This does not mean that unlinking is necessarily deterministic, however, as is explained further below, just that stochasticity is associated with the units themselves. The unlinking process requires first that two connected units have a shared, enabled (Figure 3.12) unlink transition. If this unlink transition becomes (jointly) enabled and then fires, the environment breaks the connection between the two units and fires the transitions. This completes the link, communicate, unlink cycle, illustrated in full for a sample unit in Figures 3.10, 3.11, 3.12, and 3.13.

Linking, *synching*, and *unlinking* transitions assigned to a port are disabled and do not fire unless a unit is connected at that particular port to another unit. A C/E net controller transition

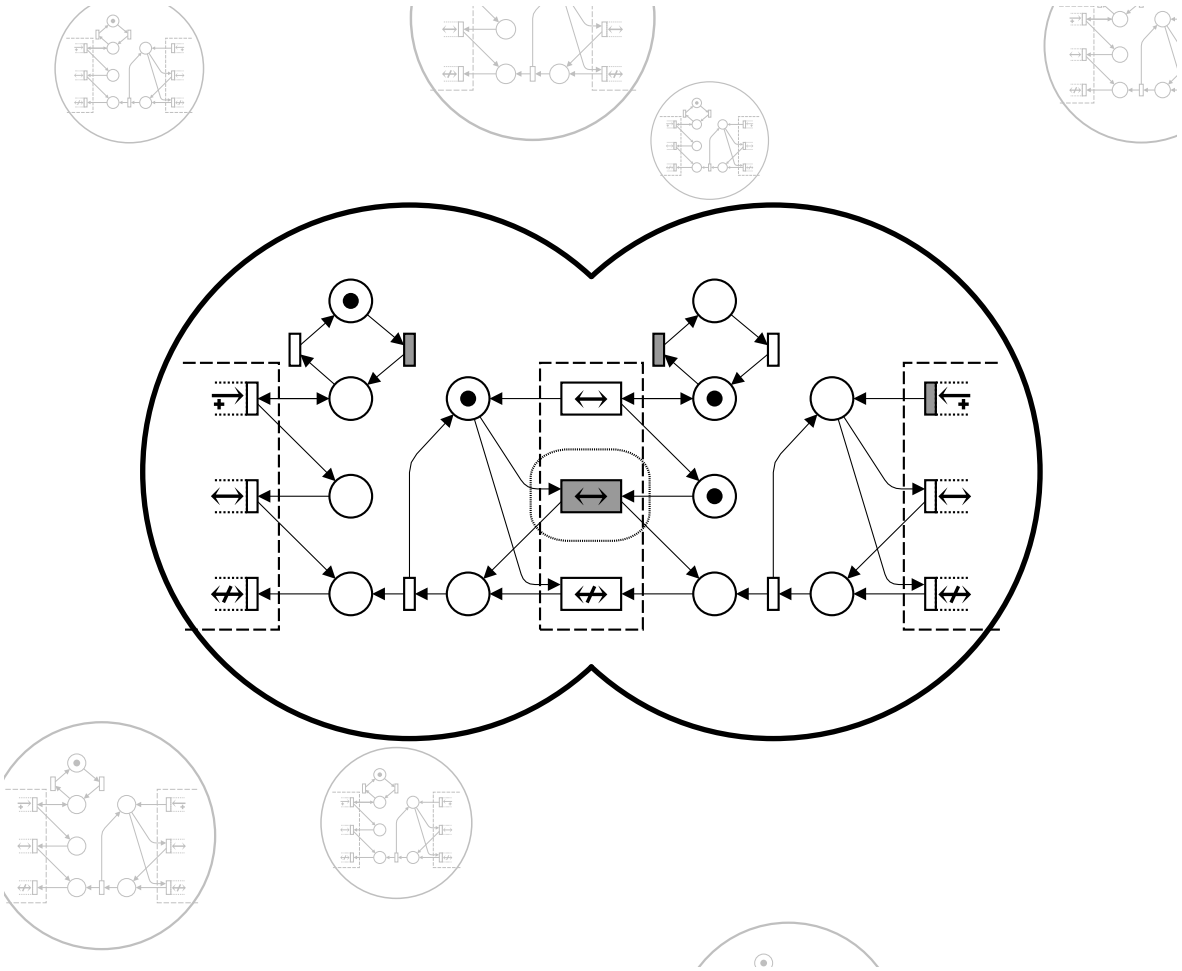


Figure 3.11: The two large units from Figure 3.10 have been assembled into a composite unit (or structure) by the environment. The assembly process fires the corresponding link transitions simultaneously, as if they were a single transition. Each transition in the linked ports is synchronized with the other corresponding transition, effectively creating single shared transitions with both left and right input and output places. The central synchronized transition is now enabled and may fire. Because they have effectively become internal transitions, the merged transitions do not need to hold for environmental input before firing.

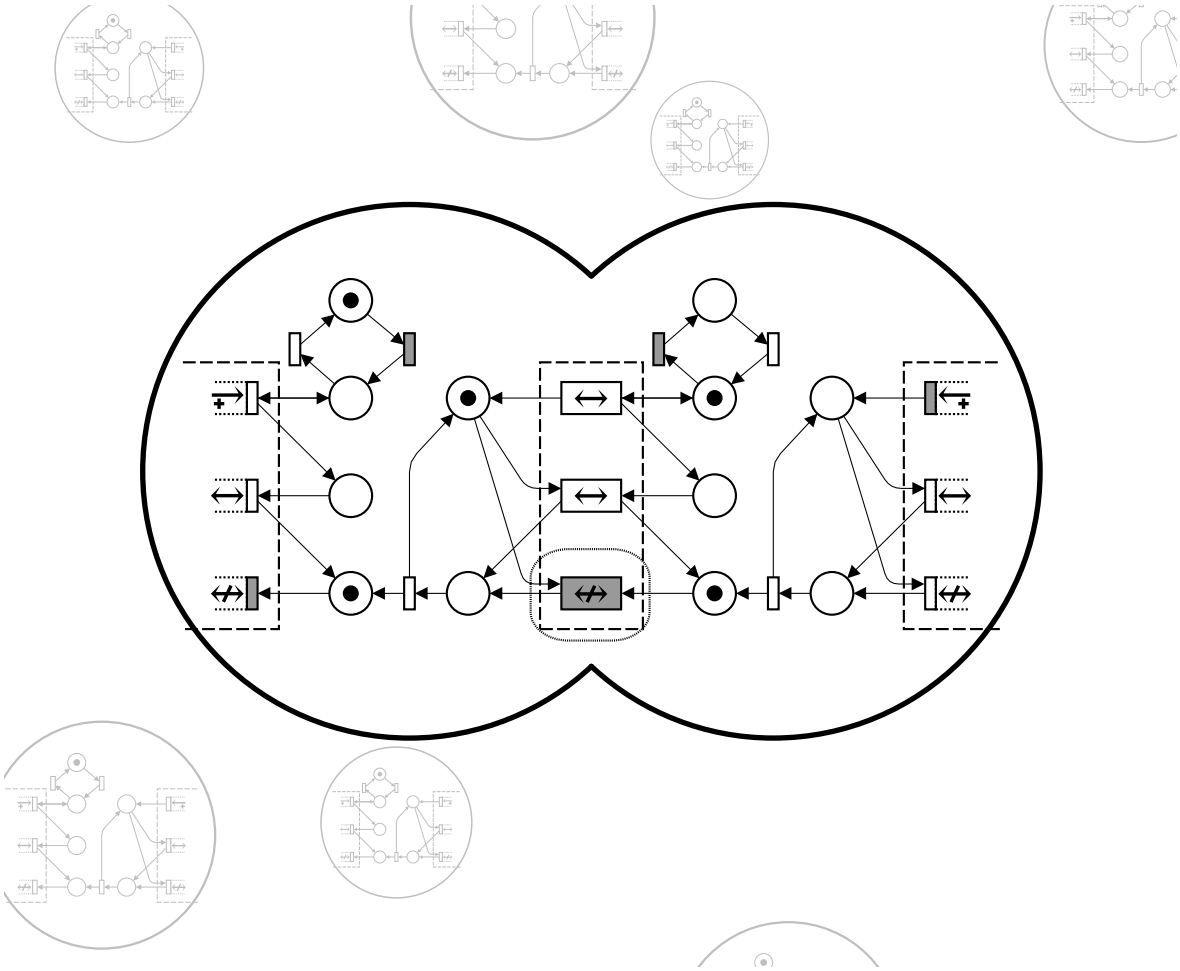


Figure 3.12: The composite unit from Figure 3.11 has fired several transitions including the central merged transition and the merged unlink transition(s) are now enabled. When these transition(s) fire, the complementary ports will be broken apart and the synchronized transitions separated.

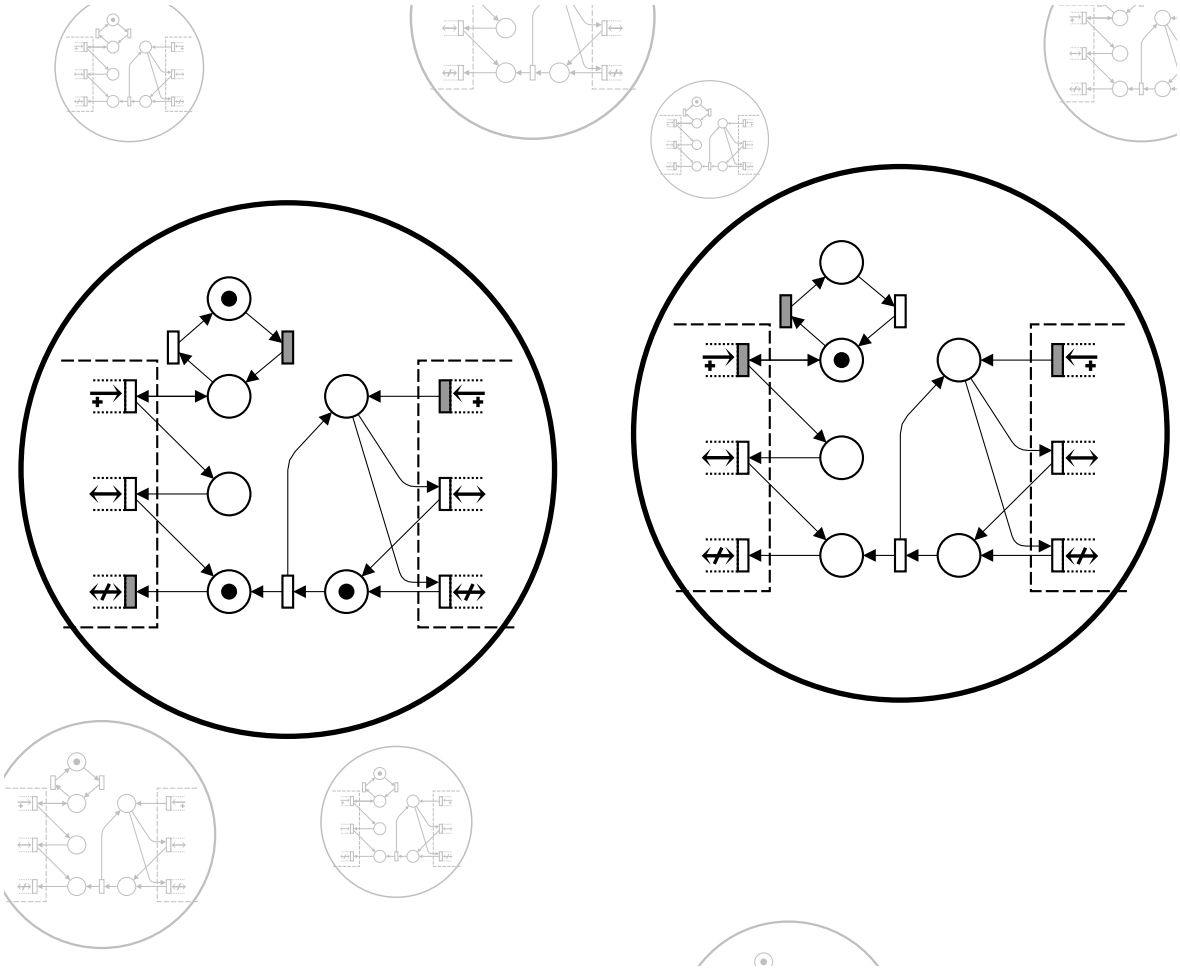


Figure 3.13: The composite unit from Figure 3.12 has now fired the unlinking transition(s) and broken apart into two separate units once again. The assembly process has changed the state of the unit on the left, which will impact future interactions. This is also an important mechanism of symmetry breaking in the CORAL model, as initially identical units can differentiate via assembly. The left unit may again become enabled for assembly on the right port once the enabled link transition becomes holding again, likewise for the right unit's left and right ports.

which has attempted to fire but is inhibited not to fire by these constraints is said to be *holding*. Holding only occurs after a transition would have fired normally, but was unable to do so due to environmental restrictions like being assigned as a port transition or signal transition. One can think of holding as a "frozen" transition firing, which may be unfrozen later through some environmental interactions or if the marking which enables the transition changes. Alternately one might imagine the environment as a large singular C/E net linking each of the many units, where the holding transitions are waiting for a token in a special environmental place.

Essentially, holding prevents the environment from directly driving unit actions via transition firings, since units must have already been capable of performing the action represented by the firing transition beforehand - the environment only restricts firing. Holding also makes atomic units more intuitive to design. It is possible to emulate the opposite effect, however, in which unlinked port and signal transitions *do not* hold, by adding a second set of transitions unassigned to ports but mimicking the synchronized transitions. When linking occurs, a special pair of places can be toggled and the inner transitions disabled while the synchronized transitions become enabled. Though the potential zero-time-step events used in this version of the CORAL model mechanism makes holding necessary for non-stochastic port activation, perhaps similar effects could be created entirely implicitly using only synchronization and more restrictive timed transitions. Holding is also used in the implementation of background signals, and so will be discussed in more detail in the section below.

3.5 Background signals

As introduced in the beginning of this chapter, background signals in the "sea" of assembling CORAL parts are detected by each unit, and may modify the units' behavior. These signals may be externally manipulated over time in order to direct the behavior of many parts into assemblies or, speaking in a different but equivalent paradigm, to modify the environment such that the units self-assemble into particular devices.

At every time step of a CORAL simulation a single type of background signal is present. This signal may change as the simulation progresses, and is the only mechanism used to pass external information into the CORAL environment. Background signals play the role of controllable parameters in a real assembling environment, be they chemical, electrical, sonic, etc.. Detection of background signals by units is implemented in a similar way to the linking transitions described above. An initial mapping of signals to unit controller transitions is defined in the environment, representing the effect a signal would have on some device. These mapped *signal* transitions are constrained to fire only if the appropriate signal is present in the environment (shown in Figure 3.14). As described above for linking transitions, a signal transition which may have otherwise fired but cannot do so because the background signal disallows it is said to be *holding*. The signal transition mapping is slightly simpler than in the case of port transitions because only individual transitions are assigned to particular signal types, i.e. each signal transition is independent from the others. Figure 3.15 illustrates the signaling process for a number of units in a sample environment.

Using this indirect method of disabling transitions, the environment and background signals never directly dictate that a unit change state or immediately assemble. Instead, a CORAL unit

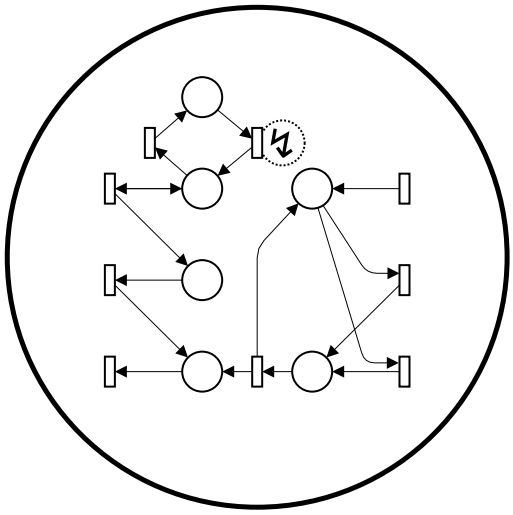


Figure 3.14: The atomic unit from Figure 3.8 with the topmost transition assigned as a signal transition, indicated by the downward zigzag arrow in the dotted circle. Multiple signal transitions are possible and used heavily by the assembling units in later chapters, but only a single signal transition is shown here for simplicity.

must be in a receptive state, indicated by holding transitions, and the environment's job is again only to execute these transitions simultaneously. Rather than increase the number of potential execution paths of the unit by adding extra functionality, the environment can only *restrict* these paths by requiring paired token or state transitions (if units are viewed as a FSM) to itself or other units. This property allows the user of the CORAL model to be confident that the environment is not implicitly introducing any extra operations which were not already present in the atomic units. Signals in the environment which do not correspond to a holding labeled signal transition are simply ignored by a unit, as shown in Figure 3.15.

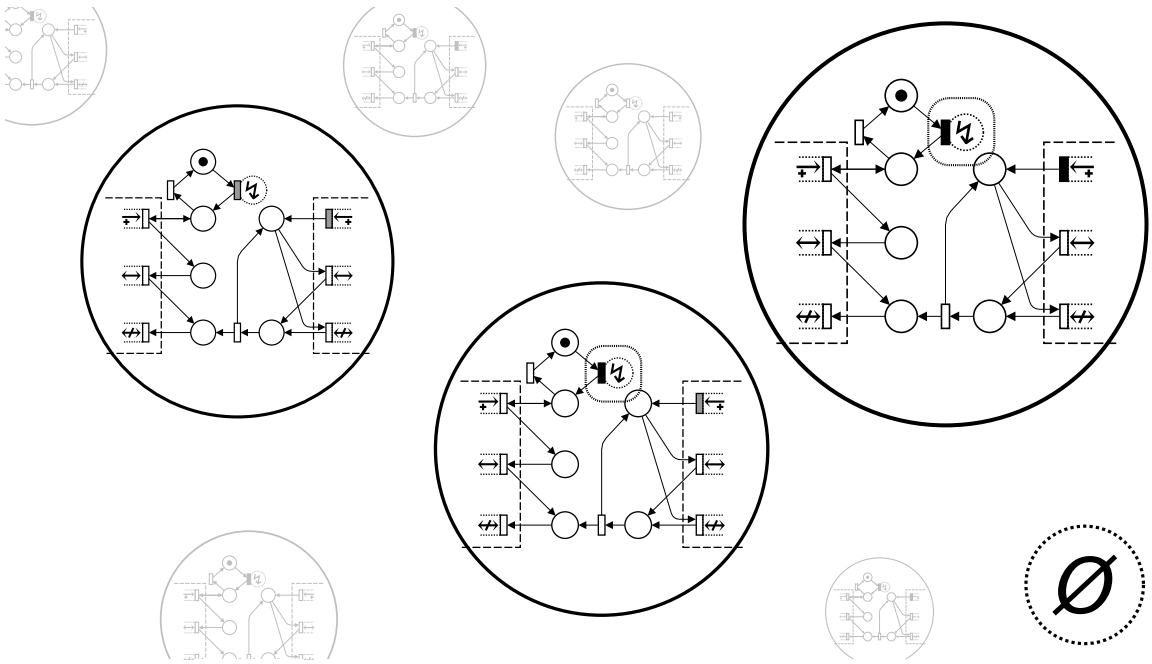
As a simulation progresses, the environmental background signal is changed repeatedly, in sequence, to stimulate different behaviors. To describe these changes compactly, a sequence of background signals sent to the environment over a period of time can be written using a shorthand syntax indicating the types of signals and the signal durations. The format is [signal]:[duration] , [signal]:[duration] , ..., where [signal] can be omitted if Ø and [duration] omitted if 1. For example, the sequence:

X:4, Y, :3, Z:2

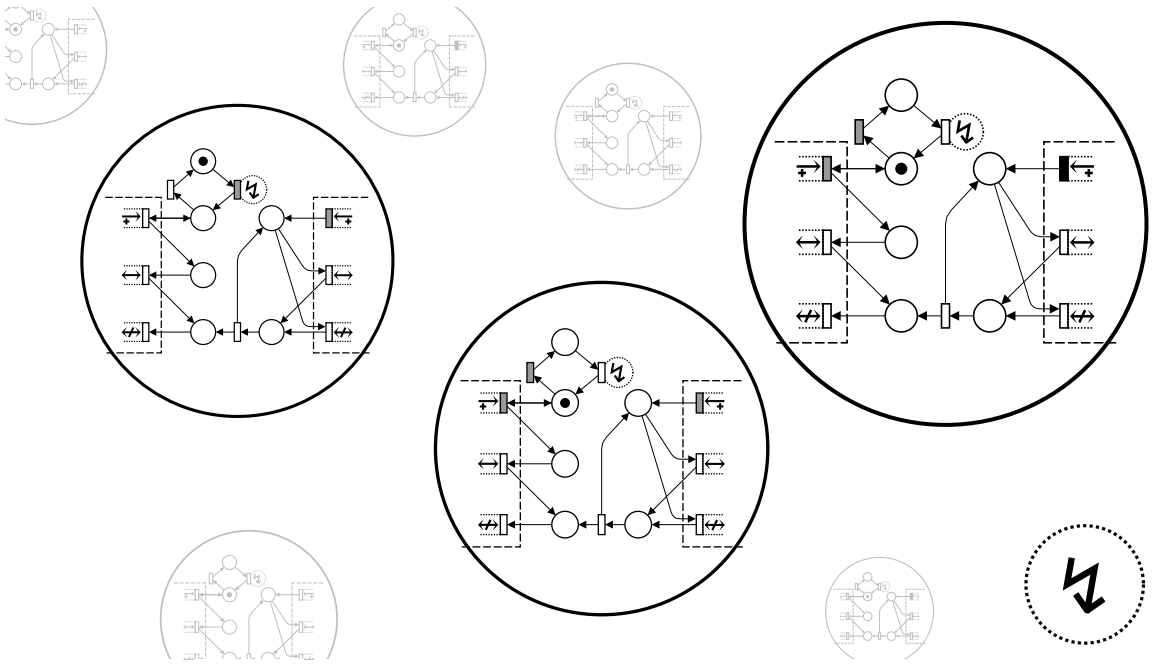
is equivalent to the expanded input sequence:

XXXXYØØØZZ

This sequence defines the background signals of some environment for 10 time steps. Shorthand is used interchangeably with the expanded notation throughout the rest of the chapters to describe background input signals set in the environment. The Ø symbol is used to represent the absence of a signal or a quiescent background signal like the quiescent states of certain cellular automata.



(a) No (empty) background signal present.



(b) At a later time, the signal changes to match the signal transition label.

Figure 3.15: Example CORAL units from Figure 3.14 before and after a background signal change. Ports are assigned as in Figure 3.9, though unused here. In (3.15a), the background signal - indicated by the large dotted circle in the lower-right corner - is empty (Ø), so transitions assigned to signals which have been enabled for long enough cannot fire and are *holding*. Note that the large unit on the left has an enabled signal transition, but is not yet holding. At a later time in (3.15b) the background signal has been changed to ↯ (again shown circled in the bottom right-hand corner), matching the label on the signal transitions, and so all previously holding signal transitions are able to fire. Units with non-holding signal transitions are unaffected. Some units now have enabled link transitions for their left ports, which after enough time has passed would allow the units to undergo assembly as shown in Figures 3.10, 3.11, 3.12, and 3.13.

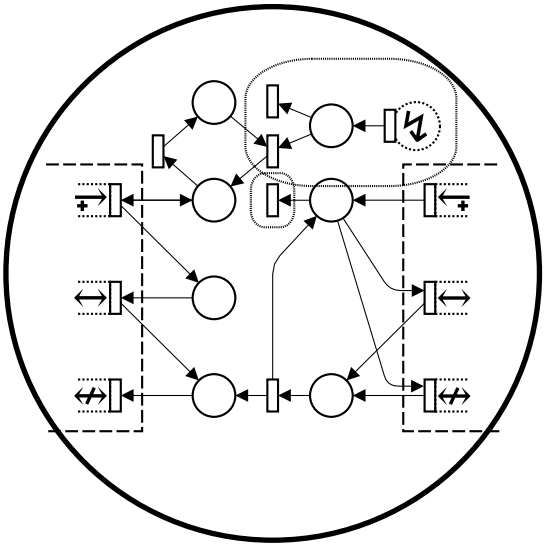


Figure 3.16: A CORAL unit similar to the ones presented in Figure 3.15 but with a construction that sometimes "forgets" it has received a signal (circled with a thinly dashed line). Additionally, this unit can become locked in the assembled position if it removes a token placed by the right linking transition needed later for unlinking (transition also circled).

3.5.1 Signal noise

Without exception, background signals are received by every receptive unit in the simulation. Similarly, when two receptive units are chosen to assemble, assembly always succeeds. In the real world, noise and errors are often present and a significant factor in designing assembling devices. These issues are important, and are not modeled in the CORAL environment directly but rather in properties of the unit C/E controllers themselves. For example, if one wished to simulate a unit which only received a fraction of the broadcasts, the C/E controller on the unit itself would be designed to nondeterministically throw away any tokens created as a result of the signal transition firing (shown in Figure 3.16). The signal transition still fires perfectly, but the unit only responds some of the time.

This approach to noise was chosen to allow flexibility in how signals are processed from each signal transition while keeping the environment very simple. Units which fail to assemble can be modeled using internal logic which sometimes immediately detaches attached units. It makes no sense in the context of the environmental assumptions to add an ability of units to refuse a connection; units cannot be affected by any other unit until assembly is performed. In the experiments described in Chapters 4, 5, and 6, noise was not added explicitly to any C/E net controller but inconsistent functioning is often observed due to differences in transition firing sequences. Much of the design in the controllers for Chapters 4 and 5 is explicitly to ensure deterministic functioning for particular actions, and generally when evolving controllers in Chapter 6 the resulting units work only stochastically. Timing issues, described below, along with incorrect execution branching, provide plenty of scope for noisy execution when viewed at the proper semantic level.

3.6 Time constants

The CORAL simulation framework has thus far been defined with the assumption that the execution of the C/E net controllers inside each unit is nondeterministic. The frequency of the assembly operation has also been left undefined. When actually simulating instances of units interacting, transition timing must be defined - or more accurately, the relationship between how quickly the transitions on the unit controller fire to the assembly operation must be defined. When this constant is chosen, the C/E controllers become timed, stochastic Petri nets (SPNs) (Peterson, 1981; Jensen & Rozenberg, 1991). Non-instantaneous internal transitions are a natural fit in the CORAL model because environmental transitions must sometimes hold, and these eventually become the internal transitions of the composite units. In addition, assuming units reach a steady state before environmental interactions become important (or that they ever do) limits the applicability of the model for non-equilibrium devices.

In the designed and evolutionary assembler work presented below, the CORAL simulations are executed discretely as a series of timesteps. Transitions, once enabled to fire, wait a random amount of time to fire, uniformly chosen from $[0, \tau)$ where τ is an integer (often 5 or 20). The potential choice of 0 as the lower bound is significant because it allows for arbitrarily long sequences of execution each timestep. During these sequences, the external agent controlling the background signals has no means of controlling the unit, defeating attempts to deterministically time when a unit will reach a particular state. The use of another gatekeeper or "clock" signal is necessary if this kind of determinism is required (though longer execution sequences become exponentially less likely). This also means that assembled units cannot easily rely on each unit remaining in semi-synchronized execution paths after a communication (i.e. synchronized transition) action. If the marking changes and a transition waiting to fire becomes inactive due to other transition firings, the transition must wait again when re-enabled. Holding transitions must also wait a random amount of time to activate, but then fire immediately when the appropriate signal or assembly operation occurs. As discussed at the end of Section 3.4, the non-holding equivalent is also possible to emulate via a simple construction duplicating the port and signal transitions.

The rate of assembly in the environment is determined partially by the number of units with open ports (i.e. holding link transitions), but is limited to α assembly operations per timestep (α is often 10). The number of operations is not dependent on the number of units in the environment or number of open ports.

The particular units and ports which undergo assembly when more than α potential operations are available are chosen randomly. In many real systems this method of matching assembling units would be very implausible; in chemical systems, for example, the rate of assembly is a function of the amounts of the components. Simplicity was chosen in the CORAL model so as not to introduce other complicating factors to the unit interactions aside from the background signal. However, extensions using different interaction types were designed to be simple modifications of these assumptions. Real assembling systems differ greatly in the way in which parts find one another, so any choice of interaction makes the simulation less general. By investigating this baseline type of extremely simple pairing behavior, particular extensions to the unit assembly choices such as locality or physical conformation can be quantified in terms of the extra expressive power it adds (or removes) toward building particular structures. Locality in particular is discussed

later in Section 5 as significantly reducing complexity when building structures with cycles in the connection topologies.

State space methods

Interestingly, there is an isomorphism between SPNs with exponential delays and Markov models (Molloy, 1982; Lin & Marinescu, 1988). More generally, any labeled transition system (LTS), to which C/E and Petri nets can be transformed, has a state-space interpretation which can be analyzed in generic ways for interesting properties (Valmari, 1998). Such state space analyses are used in Chapter 6 to algorithmically filter out Petri net controllers which can never interact with other units or input signals, as well as to understand how evolved C/E net unit controllers function.

To this end, the uniform-delay and uniform assembly rate assumptions of the CORAL model could have been replaced with exponential delays, somewhat reducing determinism but still allowing similar behavior. As the focus of the work in this thesis was to establish that scalable, computational assembling devices could be defined rigorously and constructed, further analysis of these devices was only preliminary. A simple modification of the model to use different timing mechanisms would be an interesting and useful step in that direction, and is mentioned as a future extension in Chapter 8.

3.7 A simulation example

Once the assembling unit C/E logic has been defined and environmental transitions have been assigned, the behavior of large numbers of these units can be simulated. Simulation verifies that unit designs do in fact behave as intended, as well as providing a very useful testbed for trying out and debugging new assembly ideas. Simulation is also used in particular to verify the behavior of the scalable assembling devices in the next chapter. In this next section, a simple simulation of the assembling units introduced above is presented, demonstrating the core features of the CORAL simulator and simple types of chaining behavior. As might have been guessed, the example C/E unit introduced above was designed to have interesting, though somewhat limited, assembly behavior.

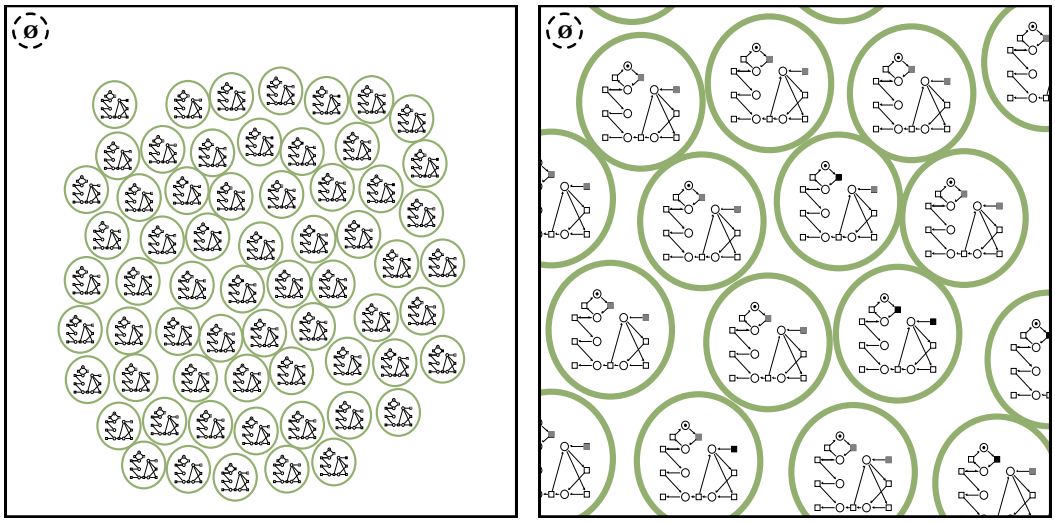
3.7.1 Sample environment

A sample simulation environment, filled with copies of the units described in the above sections, is shown as Figure 3.17. The units and environment are visualized and simulated via a slightly customized MASON agent simulation framework, developed at George Mason University (Luke et al., 2004), while the C/E net simulation engine itself and the environment interactions are programmed in Java, available to download from:

- https://coralassembly.wordpress.com/

Images from the simulation are generally shown outlined to distinguish them from other unit diagrams in the text.

In the simulation shown, the time constant for transition firing τ is set to 20, and the assembly rate α is set to 10 assembly operations per time step. As before, this means that each transition will wait in the interval of $[0, \tau)$ time steps before firing, and that up to 10 open complementary



(a) (b)

Figure 3.17: Images from an example simulated environment containing of 64 identical copies of the assembling units shown in Figure 3.15. Image (3.17a) shows all the units, while (3.17b) is a close-up view. Because it is often difficult to track the states of many units simultaneously, the external borders of the units drawn in the simulator are assigned arbitrary colors based on state - units with the same marking have the same color. All units currently have the same marking, so all colors are currently the same. As in previous figures, the current background signal broadcast to all units is shown in the upper-left hand corner of the image (Ø), but ports and transition assignments are not visualized. Instead, as is shown in Figure 3.19, assembled units are drawn with transitions linked by dotted lines.

ports will be paired each time step if they exist. The behavior of this particular unit and many others is somewhat insensitive to these constants, however, and would generate similar structure if they were changed (though more slowly or quickly).

The CORAL environment is visualized in the simulator as an empty white plain on which CORAL units float about, often bumping into one another. This bumping is purely a visualization trick to "spread" units apart when assembled into different kinds of structures, the position and movement of a unit has no influence on which pairs of units the environment chooses to assemble. When units assemble they are also depicted as linked by ideal springs, which again does not affect the simulation itself. Since the topology of assembled structures is dynamic and often impossible to predict in advance, a spring network is just a useful, general model from which natural structural layout can emerge.

3.7.2 Signal control

As was the case in Figure 3.15a, each unit in Figure 3.17 starts with only a single token in the topmost place. Since a token is required in the place below for the left port of the unit to open, no assembly can take place even though the right port link transition eventually holds, opening all right ports. The enabled signal transition cannot fire without a matching background signal, and so the units simply remain static until that signal is sent. Several time steps are allowed to go by, so

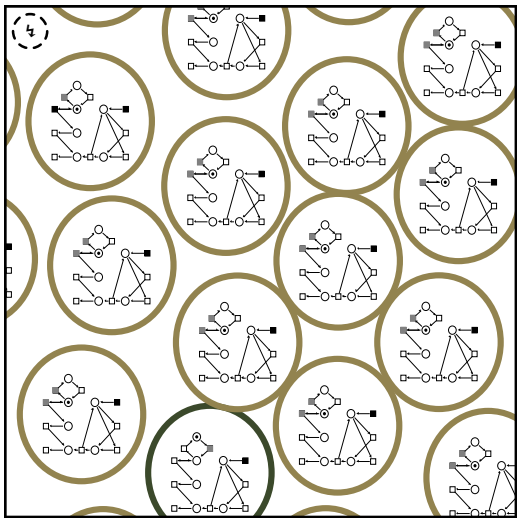


Figure 3.18: The simulation environment from Figure 3.17 after the background signal has been changed to ↯. The firing of the signal transition changes the controller markings, enabling the left port link transition. When the left port link transition attempts to fire it will hold instead, opening the left port of the parent unit. A single unit has already fired another internal transition, however, returning it to the initial state, and depending on whether the internal or link transition fires first each unit may or may not open the left port. Again, the colors are arbitrary and different colors indicate different unit markings.

that each signal transition is holding. Eventually, in Figure 3.18, the background signal changes, matching the signal transition label, allowing the signal transitions on all the units to fire.

When the signal transition fires, the new marking enables many of the left port link transitions, again as seen in Figure 3.18. Some of these transitions will attempt to fire and hold if other internal transitions do not remove the enabling tokens, resulting in the left port opening on several units. Since the right port of every unit is also open, the environment pairs these complementary ports randomly, resulting in unit structures shown in Figure 3.19. Not only pairs but also chains and loops of units are formed, since the open left and right ports of paired units may also be open. In general, these structures will not be initially stable because of the C/E controller design chosen. The first connection made is initially destroyed by the paired units, as in Figures 3.11 to 3.13. The transient assembly changes the markings of some units, however, resulting in later structures with stable parts. The simulation after several thousand time steps have passed is shown below as Figure 3.20, and includes three stable parts which have re-formed from units in previously assembled structures.

Without any other changes to the background signal, no units in the simulation are able to assemble or change state, having become locked into a particular marking. The design of the signal transition, however, allows it to fire multiple times, each time "pumping" a single token from the topmost place to which it eventually returns. As seen above, each pump potentially enables the left unit port, and it is instructive to view the simulation results if the background signal continually pushes the token around the upper loop. The units introduced later in Chapters 4 and 5 use long, repeated signal inputs as well.

Figure 3.21 shows the final steady state of the simulation after continuous firing of the signal

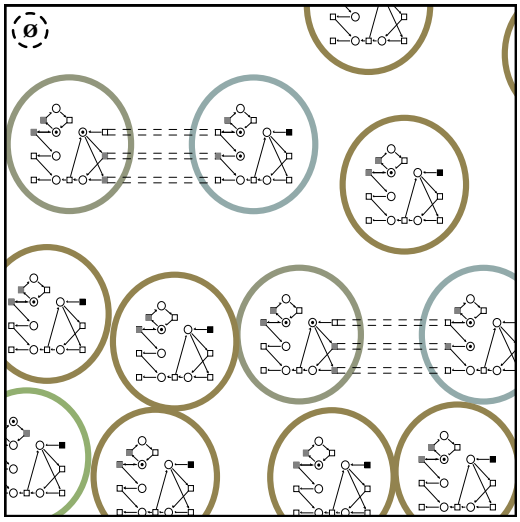


Figure 3.19: Some of the left ports were opened after the signal in Figure 3.18, resulting in several unit pairs. Two of these structures are shown above, with dotted lines linking synchronized transitions.

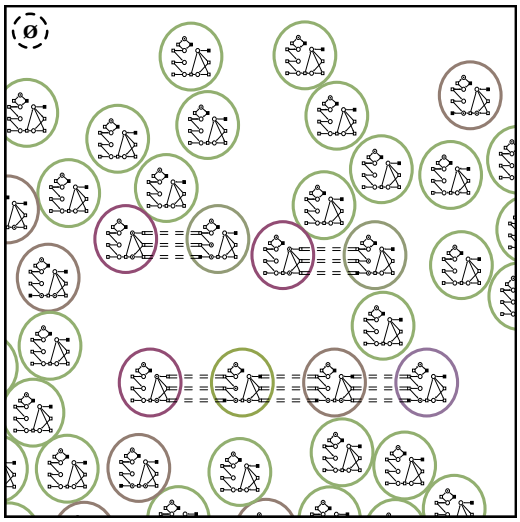


Figure 3.20: The simulation after several thousand timesteps have passed without a background signal, resulting in three stable structures. The unit markings, indicated by darker color (or brightness), are different here than in the previous paired structures of Figure 3.19.

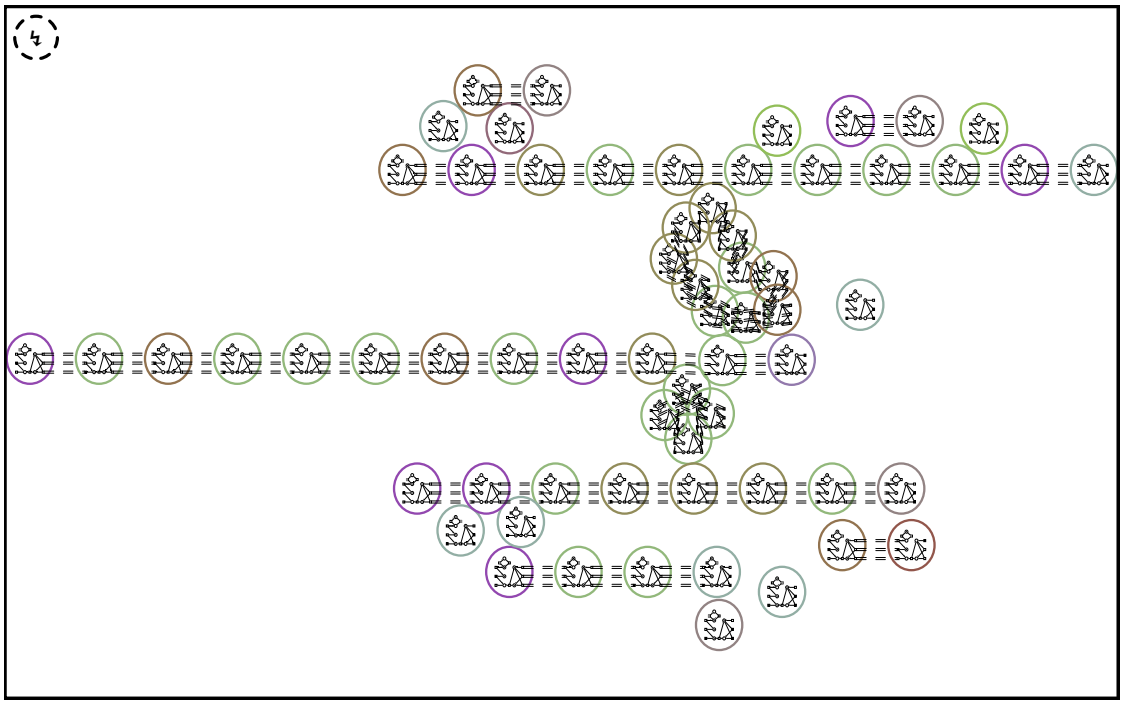


Figure 3.21: More assembly has taken place after many time steps with the ↯ background signal. Many chain structures are visible, along with two loop structures of length 4 and 10. The unit structures are topologically stable under the influence of any other background signal patterns, and therefore no further assembly can take place.

transition. A variety of chain structures form, of different lengths, and eventually each of these becomes stable to further assembly - though of course the token pumping cycle in the topmost places continues. The intuitive reason for the structural stability is that repeated assembly operations result in additional tokens in the leftmost places of a right-port-assembled unit. These extra tokens eventually cause contact, or "jam," the left port transitions, resulting in assembled units which cannot disconnect or connect depending on the current assembled state. The resulting structures are formed stochastically, stopping growth and breakage when all the composed units are unable to disconnect or attach to others. Because there is only a single complementary pair of ports, only chains or loops are possible when this occurs.

3.7.3 Past steady state?

This example simulation was designed to give a flavor of the types of behavior seen in assembling CORAL units, and the static final state is typical of most kinds of C/E net controllers. Eventually, no matter how the background signals are manipulated, these controllers reach a final set of steady state structures from which no new interactions can be generated. Different structures may emerge depending on the random choices of transition firings, or dynamically stable organizations, but the final result is effectively bounded in both size and complexity. Analogous behavior is seen in real-life robotic assembling devices, where a particular type of controller is sufficient for building a certain structure design or set of structure designs and after which the controller must be reprogrammed. This type of interaction requires a huge unit memory, and raises the question of whether there might be another, developmental way to *reprogram the structures themselves*

through the same types of signals.

As mentioned in Chapter 2, it seems the natural world currently has a monopoly on these type of assembling systems which do continue to generate interesting behavior across scales - nucleotides and amino acids effectively assemble into dynamic structures vastly larger than their size. Admittedly, each organism does not bootstrap itself from only information, but the question is still valid: is there a small unit from which one can build things vastly larger and arbitrarily complex, using only information passed through a highly limited channel? The CORAL model, by disallowing all but a single form of external input, makes this question particularly stark. Answering this question is the topic of the next three chapters, which give constructions for an infinitely scalable logical assembler and finite assembly device able to assemble into arbitrarily complex computing devices. Chapter 6 takes a different approach, and uses evolutionary search to directly investigate the idea of multi-scale assembly.

Chapter 4

Assembly Scaling with NOR Operations

As discussed in Chapters 2 and 3, current research into assembling systems and self-assembling artifacts has not, to date, focused on assembly as a process divorced from scale. Structures of many components are often fundamentally different in some way from the units themselves, and when composite units are not different, they are instead functionally abstract and difficult to map into real-world interactions. In the previous chapter, the CORAL model of assembling systems was introduced which address these two constraints, expanding a precise definition of limited-mass assembly to interacting units of any size and introducing a typical assembler which is limited in dynamic and computational function. In this chapter, we use the CORAL model as a platform for designing a novel assembling unit which is capable of building *itself* at larger levels, as an explicit demonstration of an assembly process which scales, controllably, indefinitely. Designed to be as simple as possible and driven by background environmental signals, the assembling Not-OR (NOR) unit in structural combination can emulate any type of logical expression. This is partially a function of the inherent composability of logical operations and partially of the scalable assembly process presented here, allowing the composition to occur no matter how large structures grow. With indefinite hierarchical assembly, structures built from huge numbers of finite-memory NOR units can perform well-defined operations using vast numbers of parts. The scalable logic operations are here presented for assembly tasks only, but by linking these operations to other real-world actions the coordinated behavior of huge assemblies becomes possible.

4.1 Not-OR operation

It is useful to begin by introducing some basic propositional logic, as the NOR units defined further in the chapter rely on it heavily. The quick introduction below is largely derived from (Magnus, 2009), though (Smullyan, 1995) was also a resource. Propositional logic consists of an infinite set of *variables* or atoms (written here as lowercase alphabetical letters) to which *true* or *false* values can be assigned, along with a set of logical *operators* which combine the values of these primitives. Most of the standard operators are quite familiar, like AND ($\wedge$), OR ($\vee$), IF (conditional) ($\rightarrow$), and NOT ($\neg$), though the particular symbols used to represent these operators varies. NOR ($\downarrow$) is another basic binary logical operation, perhaps not as common, but is defined

simply by the truth table below:

a	b	$a \downarrow b$
true	*true*	*false*
true	*false*	*false*
false	*true*	*false*
false	*false*	*true*

Table 4.1: NOR truth table.

As can also be seen in the above table, the output of NOR is the negation of the OR of two inputs, i.e. $a \downarrow b = \neg(a \vee b)$. Essentially, the output of a NOR operation is *true* only when both variables (or atoms) a and b are *false*. Combinations of NOR operations can mimic any other binary logical expression (Smullyan, 1995), a property which is shared by another logical operation NAND and used when designing transistor logic in microchip design. For example:

$$\begin{aligned} \neg a &= a \downarrow a \\ a \vee b &= (a \downarrow b) \downarrow (a \downarrow b) \\ a \wedge b &= (a \downarrow a) \downarrow (b \downarrow b) \\ a \rightarrow b &= (\neg a) \vee b \;=\; ((a \downarrow a) \downarrow b) \downarrow ((a \downarrow a) \downarrow b) \end{aligned} \tag{4.1}$$

The identities from Equations 4.1 can easily be proven by substituting all combinations of *true* and *false* into the variables a and b for each equation, or equivalently by constructing truth tables. These expressions can also be represented using diagrams:

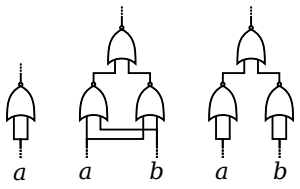


Figure 4.1: The first three identities $\neg a$ (4.1a), $a \vee b$ (4.1b), and $a \wedge b$ (4.1c) from Equations 4.1, shown as a diagram. Each NOR node, represented traditionally as an extended half-moon with circled tip, is linked so that input enters from the bottom and output is read from the top. The specified a and b variables are set at the base of the diagram and outputs read from the topmost NOR node. Each subfigure corresponds to the respective equation from the identities.

NOR operations can also be combined into a larger expression or diagram which also computes a NOR function, as shown below in Equation 4.2 and Figure 4.2:

$$((a \downarrow b) \downarrow (a \downarrow b)) \downarrow ((a \downarrow b) \downarrow (a \downarrow b)) = (a \downarrow b) \tag{4.2}$$

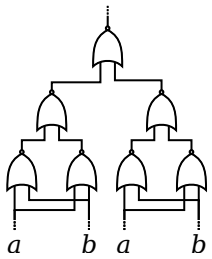


Figure 4.2: Diagram of nested NOR gates which also computes NOR.

Using this construction, the NOR operation supports some notion of infinite hierarchical assembly (as is well-known). When combined or "wired" with itself in particular configurations, the output of the NOR-structure in response to some input is identical to the output of individual NOR units. These larger combinations can themselves be wired together to create other logical expressions using arbitrary numbers of units. The mapping between units at different scales is not entirely well-defined, however - there are four times as many inputs in the larger NOR unit. There is also no description of the process through which a single NOR unit might link in a distributed way with other NOR units - as one might expect, since propositional logic does not deal with such issues.

In the next section, a construction is presented using the CORAL model in which assembling units are able to perform NOR operations, creating a distributed implementation of the above abstract logical construction. The largest challenge is to address the concerns raised above regarding the creation of a distributed process to build logical trees. Two sets of background signals are introduced for this purpose: one set to encode the potentially complex logical inputs into a sequential form appropriate for the limited CORAL bandwidth, and another to encode assembly commands. Once constructed, the unit trees have useful scale-independent properties which are used as the basis for further assembly as discussed later in the chapter.

4.2 NOR assemblers

NOR operations are a useful primitive around which to design a scalable assembling unit, since the logical operation of meta-structures is well-defined and in combination the operation is logically universal. The processes required to control and manipulate these primitives while construction is taking place must still be specified, along with some mechanism for setting the values of different variables in NOR expressions. The CORAL model provides a natural framework to address these issues. Units and meta-units have identical internal and external interactions, so controlling the behavior of assembled structures of any size is done identically. In theory, this property allows a single encoding of background signals to be used to specify input variables and control the creation of new structures representing new logical expressions, no matter how large the structures grow. The trick, of course, is to design signals and units (from here on called NOR units) which respond in a consistent way when assembled into meta-structures. Otherwise one has to create separate background signal encodings for each of the exponentially many different structure types, defeating the purpose of investigating assembly at multiple scales.

As can be seen in the network topology of the logical diagrams Figures 4.1 and 4.2, connected NOR operations are depicted as linked to three other operations along two input "wires" and one

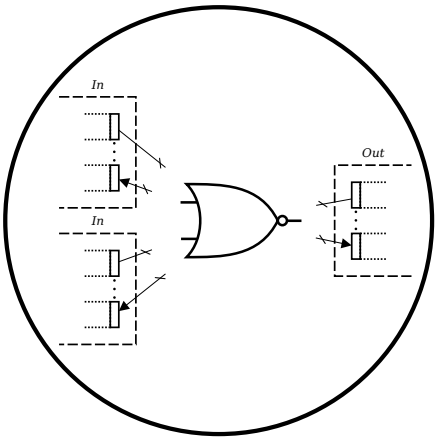


Figure 4.3: A CORAL unit with three ports, two of type *In* and one of type *Out*, each port containing some numbers of transitions. The *Out* port type is complementary to the *In* port type. Transitions and edges are shown as connecting to some as-yet-undefined controller in the empty center which emulates the NOR logic.

output wire. In the CORAL model, all communication and structure between units is achieved through the pairing of complementary ports, not wires, and so to create similar nested forms it is necessary for each unit to contain two input ports, both complementary to a single output port. The two input ports are from here on described as *In*-ports and the output port *Out*-port. Figure 4.3 shows this idea visually.

The ports of Figure 4.3 give NOR units the potential to receive, process, and output information, but it is impossible given the constraints of a CORAL environment to specify particular input to particular units (as was done in the NOR diagrams of Figures 4.1 and 4.2). Some additional method must be defined to encode variables of NOR expressions into CORAL background signals. In the approach used here, *A* or *B* signals (or both, sequentially) are sent to the simulation environment (followed by a *C clocking* signal), and these are interpreted as *true* values for the respective inputs of the $a \downarrow b$ NOR operation a NOR unit is emulating. The choice of signal names correspond to the canonical variables for a two-input logical operation, i.e. a and b. For a given NOR unit, one can arbitrarily assign an *In*-port to each of these variables a and b, so that the values of these variables are received as input from other connected units at the *A In*-port and *B In*-port, and $a \downarrow b$ is then passed as output to a forward-connected unit. Initially, however, no ports are connected, and in general there must be a way in which units receive input from background signals for controllable assembly to be possible. The solution chosen here is for each NOR unit to be metaphorically "wired" to these shared background signals when not attached to any other unit at the appropriate port.

Re-stating the ideas above, when a NOR unit "floating" in the CORAL environment (not linked to any others) receives an *A* signal followed by a *C* clocking signal, the yet-to-be-defined internal controller emulates a NOR operation $a \downarrow b$ where $a = true$. *B* signals similarly set $b = true$. The absence of an *A* or *B* signal before the *C* signal is, by default, interpreted as a *false* value for a and b. However, if the unit *is connected* to another unit via an *In*-port, the corresponding *A* or *B* background signal is ignored by the unit and the output value of the connected unit used instead

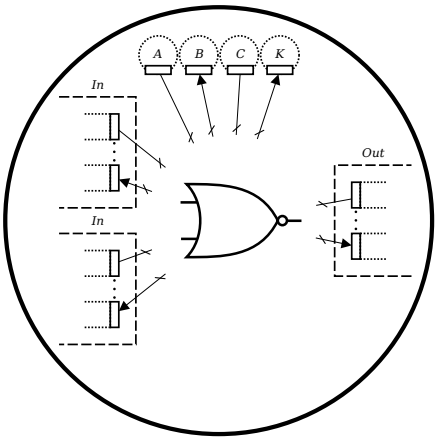


Figure 4.4: The above CORAL unit template from Figure 4.3 with additional signal transitions needed to encode logical variables. Again the rest of the controller is left undefined.

as *a* or *b*. The *A In*-port, when connected, causes the *A* signal to be discarded, and likewise the *B In*-port blocks the *B* signal (in figures, the *A In*-port is usually left or top). The overall effect is that in tree-shaped assembled structures, leaves and open *In*-ports of the tree respond to signals while fully *In*-port-connected units do not. This property mirrors the same behavior in logical operations, where atomic variables define the innermost nested level of the formula syntax (which can be viewed itself as a tree) and are where input values are specified.

Sequential, clock-separated signals like the ones described above must be used to encode the value of variables in the CORAL model, since the background signal can only be a single value at any given timestep. While a single unit might perhaps respond to all combinations of inputs as separate signals, e.g. including the combined signal *"A&B*", the total number of potential variables in a logical formula is unbounded and therefore must be specified serially at some point. The gatekeeper or clock signal *C* is also necessary because several consecutive events may occur in a single simulation timestep (see Section 3.6), allowing no deterministic way of "timing" how many signals have been sent using transition firings. For the same reason, an additional reset signal *K* is needed to restart the detection of *A* and *B* signals, as in *AB[ClocK]*.... Without a final *K* signal, there is no way to distinguish between the signal strings *ABC*, *ABCC, ABCCC*, etc. since the signal transitions synchronized with *C* may fire multiple times before the next environmental timestep. The final signal encoding sends either strings of *CK*, *ACK*, *BCK*, or *ABCK*, specifying the four *true/false* input possibilities for the NOR operation. Handling more than two input variables is also possible, as mentioned above, and discussed later in Section 4.5.

Figure 4.4 is a diagram showing the basic ports and signal transitions needed to define controllable NOR unit binary tree structures, analogous to NOR logic expressions. Structures with cyclic connections are also possible as was demonstrated in the example from Chapter 3, but these are not discussed in this chapter because they do not easily correspond to static logical expressions and are avoided in the particular assembly process introduced in the next section. The two *In* and single *Out*-ports allow connections between NOR units, while the four signal transitions allow successive signals to set the value of logical variables.

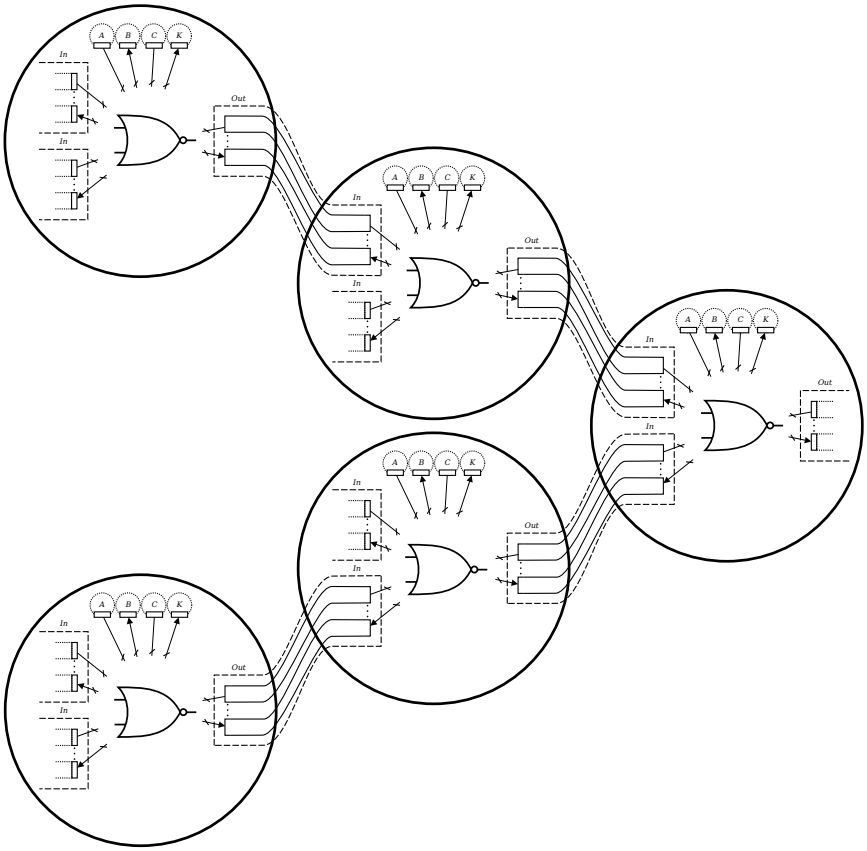


Figure 4.5: Several NOR units from Figure 4.4 connected together into a larger meta-unit. Connected ports and synchronized transitions are depicted as in Figure 3.12 but are curved to better show the structure.

4.2.1 NOR meta-units

Somehow, though the actual C/E controller has not yet been shown, the unit shown as Figure 4.4 combines the input from the two potentially-linked *In*-ports with the signals set in the environment to produce a NORed output at the linked *Out*-port (after receiving a *C* signal). When such units are built into structures such as the one in Figure 4.5, the structure has output identical to a single NOR unit, but shifted in time by an additional two *CK* signals. These larger NOR structures are referred to as meta-units. Meta-units may be grouped in the same way to create meta-meta-units since they perform the same NOR operation (Figure 4.6), and so on *ad infinitum*. Each level of structure simply delays the output by an additional factor of three, such that a level n structure requires 3^n clock signals.

The calculation of output in assembled structures and meta-units, which is where unit and meta-unit self-similarity is demonstrated, is best visualized as a wave of calculation from leaf to root (see Figure 4.7). If, for example, background signals of the form *CK*, *ACK*, *BCK*, or *ABCK* are specified as the environment advances in time, the output of a single NOR unit will be *true*, *false*, *false*, or *false*, respectively. The value of this output does not affect the behavior of the calculating unit directly, but it is fed as a value into the *In*-port of a connected unit if one exists. The output of the meta-unit shown in Figure 4.7a is initially undefined after one clock step (if the output of a structure is defined as the output of the root of the structure tree) since the background

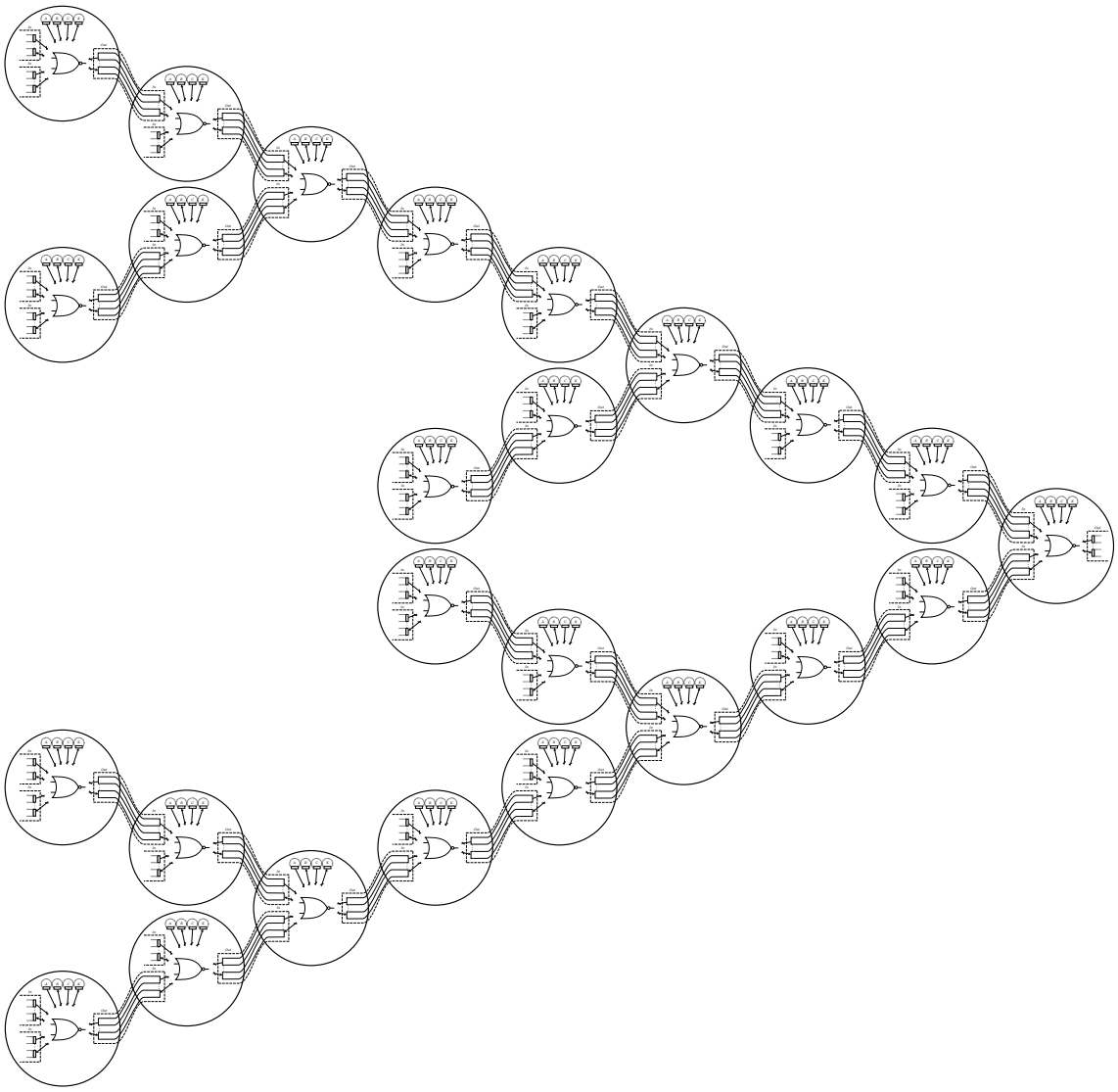


Figure 4.6: NOR meta-units from Figure 4.5 themselves connected into a larger meta-meta-unit.

signals of *A* and *B* are ignored when NOR units are connected at the corresponding *In*-port. The root of the tree is fully *In*-port connected and the previous output value of the connected units is not specified, so it is not clear what inputs the root and root's immediate children receive. If an empty background signal *CK* is then sent, assigning both *a* and *b* input variables to false on units with open ports, the root childrens' output becomes known, since the previously sent signals are known. After a second empty *CK* signal, the output of the root unit is known, and is identical to the original output of the individual unit by the identity $((a \downarrow b) \downarrow false) \downarrow (false \downarrow (a \downarrow b)) = a \downarrow b$. Every floating NOR unit *without In*-port-connected children after the *CK* signals has an output of *true*, because without connected *In*-ports a single unit simply reacts to the inputs last sent. By forming structures, inputs can instead be processed through several logical operations. Again, Figure 4.7 shows this processing graphically for a meta-unit with an initial input of *ACK*.

In general, all NOR units connected in tree structures process information this way - discrete steps starting from the leaf units and working upwards or rightwards toward the root. Each *CK* signal advances the calculation one step further upward until the final output is calculated in the root unit. Units connected at only a single *In*-port are designated *control units* because it is possible to either block or compute with the input from the connected *In*-port by sending background signals at the appropriate time. If only an empty *CK* signal is sent, indicating $a, b = false$, the value from the *In*-port connected unit passes through as if it was connected at both ports:

$$input \downarrow false = false \downarrow input = input \downarrow input$$

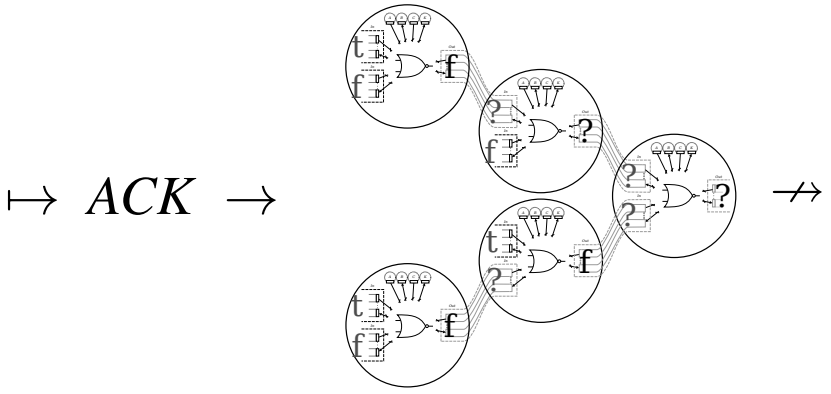
This is the reason the NOR meta-unit in Figure 4.5 differs in structure from the NOR diagram in Figure 4.2; the empty signals remove the need for the extra leaf units. Conversely, if an *A* or *B* signal is sent to the control units, indicating $a = true$ or $b = true$, input is effectively blocked from the other *In*-port since the output of the control unit is forced to *false*:

$$input \downarrow true = true \downarrow input = false$$

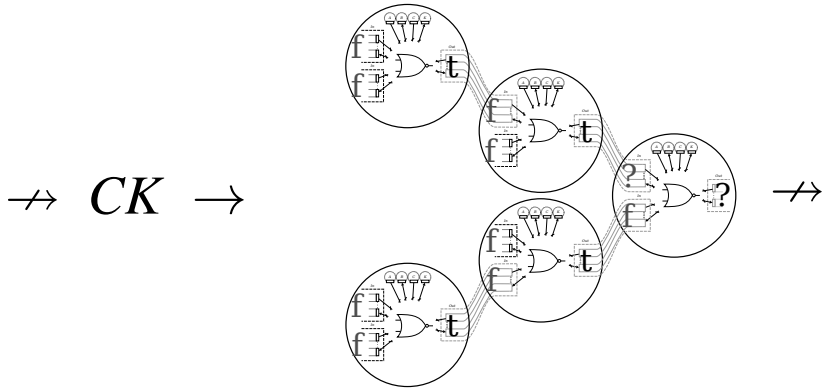
4.3 Assembly behavior

The previous section was an overview of the logical processing of NOR units and binary tree structures, which is largely identical to the conceptual NOR operation but with the added complications of signal encoding. Since the CORAL model allows only a single background signal to specify the potentially many variables of NOR operations, one must use the particular order or timing of these signals to encode the values of each variable. Once assembled into particular NOR unit structures, these structures share the recursive identity properties of the original NOR operations, albeit not in a synchronized way. This difference turns out to be very useful when distinguishing units of different sizes. However, no description has yet been made of the mechanism by which NOR units form structures, aside from the ports necessary to do so. In this section we describe a second set of additions to the NOR unit which allow us to build trees of units in a completely controllable way.

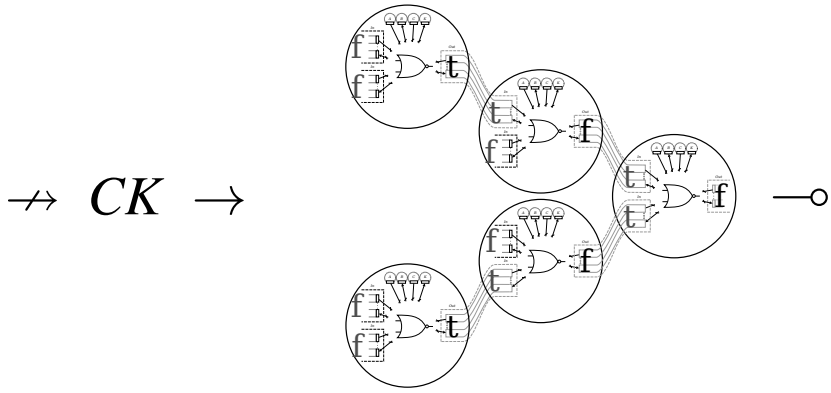
As is demonstrated by the variety of different assembling systems mentioned in Chapter 2, many distributed assembly behaviors can be imagined which allow us to build arbitrary shapes. The primary goal here of building assembling units controllable at all scales requires this behavior to survive in a meaningful way under the composition of units - in particular, the composition of



(a)



(b)



(c)

Figure 4.7: An illustration of logical values propagating through the meta-unit of Figure 4.5. The diagram is read left-to-right, top-to-bottom, where broken arrows indicate the next line. At some time in the environment the signals *ACK* are sent (indicating $a = true$ and $b = false$ at open ports), followed by two sets of all-*false* *CK* signals. At first the outputs of the rightmost units are undefined since the connected inputs could previously have passed any truth value, but as successive *CK* clock signals are set the defined calculation moves forward. After three *CK* clock steps, the root output of the meta-unit is *false* - a NOR operation has been performed on the original input signal of *ACK* ($true \downarrow false = false$).

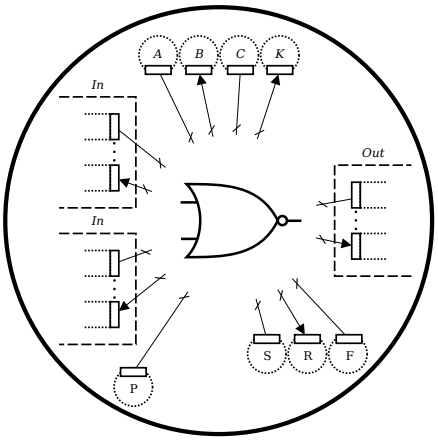


Figure 4.8: The NOR unit from Figure 4.4 with additional assembly signal transitions.

units into the larger meta-structures. If an entirely new behavior has to be used each time a larger NOR structure is formed as in (Rus & Vona, 1999; Kotay & Rus, 2000; Detweiler et al., 2007), where possibly such a behavior does not exist or requires new unit memory or functionality as in (Rasmussen et al., 2001b), it will defeat the purpose of scalable assembly by making larger structures increasingly complex or perhaps impossible to create using simple parts. The algorithm for NOR unit assembly avoids these pitfalls, and is identical (aside from a scaled timing factor) when applied to individual NOR units or meta-units or meta-meta-units of any size. NOR units themselves require only the limited behavior described in the previous section along with a small set of additional assembly signals.

A core feature of the CORAL model is that every unit is completely identical except for the internal state contained in the controller marking. For NOR units, the only state implied so far is the output state of a unit, which is affected by the background signals and potential input from connected *In*-ports. This can be seen most clearly in Figure 4.7, where it is represented by the *true*, *false*, or unknown output at each unit's *Out*-port. By adding extra signals *S*, *R*, *F*, and *P*, this state can do double-duty; it both stores the output passed to connected units and also controls the connection state of the units' *In* and *Out*-ports. Since these output states store the result of logical expressions emulated by the current unit structures, and the inputs to these expressions are controlled through background signals, this allows the control of assembly *based on current logical state*. Four new background signals are used to take advantage of this idea:

- The *S* signal (for *structure*) allows a unit to be *primed* for assembly if the current output state is *true*.
- Once primed, the *R* and *F* signals (for *reverse* and *forward*) determine whether the *Out*-port or one of the *In*-ports are to be opened or enabled. There is only a single *Out*-port, so the *F* signal directly enables the *Out*-port link transition, but the choice of *In*-ports requires an additional signal.
- A priming signal *P* "pushes back" the priming after the *R* signal sets the direction. This either opens an *In*-port which is not connected or sends the priming backward down the tree to a child unit. The selection of *In*-port is determined by the current output state of the unit, where the *A In*-port is opened by *true* and *B In*-port opened by *false*.

Overall, assembly is realized by 1) sending *A* and *B* signals to generate a *true* output in particular structures, 2) sending an *S* signal to prime, 3) sending an *R* or *F* signal to choose assembly direction, and, if needed, 4) sending additional priming signals to select child *In*-port. Figure 4.8 shows a NOR unit with these signal transitions added.

This assembly mechanism reduces building particular structures to a process of placing particular units in *true* output states. These output states are only dependent on the structurally-defined logical operations, which as shown above can be recursively composed. In consequence, the procedure for placing units in these states also remains unchanged when the structures are built from meta-NOR or meta-meta-NOR units. The full algorithm, *waterfall selection*, is described in detail below, but first requires two core techniques be introduced:

- *blinking* - a scale-independent synchronization of NOR structures and meta-units using alternating background signals
- and *shielding* - the enforced ignorance of *A* and *B* background signals in meta-units which are connected at particular *In*-ports, yet still have many other open but unused *In*-ports

Given the tools of blinking and shielding, waterfall selection can select particular structure sizes and ensure that only those structures are activated for assembly. The next two subsections will formally describe these properties and show how they can be derived for linked units emulating NOR operations. The third subsection brings the ideas together for the final assembly algorithm, which exploits the natural recursive combination of logical operations to allow the natural recursive assembly of *objects*.

4.3.1 Structure blinking

Binary tree structures built from NOR units have an interesting property - if an oscillating signal of either input variable is received by unit structures, the output state of every NOR unit in the environment will eventually oscillate out-of-phase with the signal, no matter how the unit is connected. Out-of-phase in this context means that after an *ACK* or *BCK* background signal has been present, indicating $a = true$ or $b = true$, the output state of the root of every NOR structure will eventually (after many oscillations) always be *false*, and when the signal is only *CK* the root output will be *true*. The idea may seem counter-intuitive at first since it applies regardless of NOR unit tree shape, but is easily shown using a recursive proof.

Definition 1. An oscillating background signal of *A* is defined as a series of repeating signal groups of the form *ACK CK ACK CK ACK CK*..., consisting of *(A* and clock signals) alternating with (just clock signals). Oscillating background signals of *B* are defined similarly.

Definition 2. Given an oscillating background signal, an *out-of-phase* output is defined for a tree structure when that tree structure's root has a *true* output after a group of *CK* signals and a *false* output after a group of *ACK* or *BCK* signals.

Theorem 3. The root output state of a NOR unit or tree of connected NOR units of any depth $d > 0$ will *eventually* oscillate out-of-phase to an oscillating signal of either *A or B,* given enough alternations of the signal groups.

Case 1. *A single, depth $d = 1$, non-connected NOR unit's output oscillates out-of-phase with an oscillating signal of either A* or *B*.

The above is true, because after a NOR unit receives the group of background signals *ACK* (representing $a = true, b = false$), by the functionality assumed for NOR units in Section 4.2 the output state of the NOR unit will be *false*. When *CK* are the last signals received (representing $a = false, b = false$) the output will be *true*. The same holds for oscillating *B* signals.

Case 1. *Assuming that the theorem holds for depth $\leq d$, where $d \geq 1$, the root output state of a tree structure of NOR units of depth $d+1$ will eventually oscillate out-of-phase to an oscillating input of either A* or *B*.

The root of a tree structure of depth $d+1$ has, by definition, at least one child tree of depth $\leq d$ connected to *In*-ports. The input to a NOR unit from each connected *In*-port is the output of the root of the attached child trees at the previous clock step, as described in the previous section, and so by the recursive assumption we can say that the previous output state of the child tree roots must eventually oscillate out-of-phase with the background signals after enough signal group oscillations.

In addition, by the assumed functionality of NOR units, the root NOR unit of a structure will have an output after the next signal group which is the NOR'ed input of connected child NOR units (if they exist) or the values assigned by the next signals. Furthermore, the value of the input of child units is the output calculated after the previous signal group, since calculation takes place in a NOR unit after a *C* signal is sensed.

After the sufficient number of oscillations, if the next signal group is *CK*, the previous signal group must have been *ACK* or *BCK* (see Figure 4.9). By the operation of NOR units, the next *CK* signal sets the corresponding input of any open ports of the root unit to *false*. We also know the input received from connected child tree roots must also be out-of-phase with *ACK* or *BCK*, and so is also *false.* Therefore, no matter how the root unit is connected to child trees, the output of the root unit after the *CK* signal group will be $false \downarrow false = true$.

If the next signal group is *ACK* or *BCK*, the previous signal group must have been *CK* (again see Figure 4.9). Given this, we know as before that the out-of-phase child input(s) to the root NOR unit must be *true*. Since at least one child subtree must exist (otherwise the depth of the tree structure would be 1), the output of the root unit after the *ACK* or *BCK* signal group will be $true \downarrow x = x \downarrow true = false$.

Because the next signal group is always either *CK*, *ACK*, or *BCK* when assuming alternating *A* or *B* signals, there are no other cases and the output of the root of the structure will always be out-of-phase with the next oscillating input (given at least enough alternations of the signal groups for the child trees to become out-of-phase and an additional alternation).

Proof. Since the root output state of a $d = 1$ NOR tree eventually oscillates out-of-phase with oscillating *A* or *B* signals, and the same applies in the recursive case for $d+1$ when the recursive assumption holds for $d \geq 1$, NOR binary tree structures of all depths eventually oscillate out of phase to oscillating signals. □

Since every NOR unit in a binary tree structure of any size is the root of a tree of some depth, this statement is equivalent to:

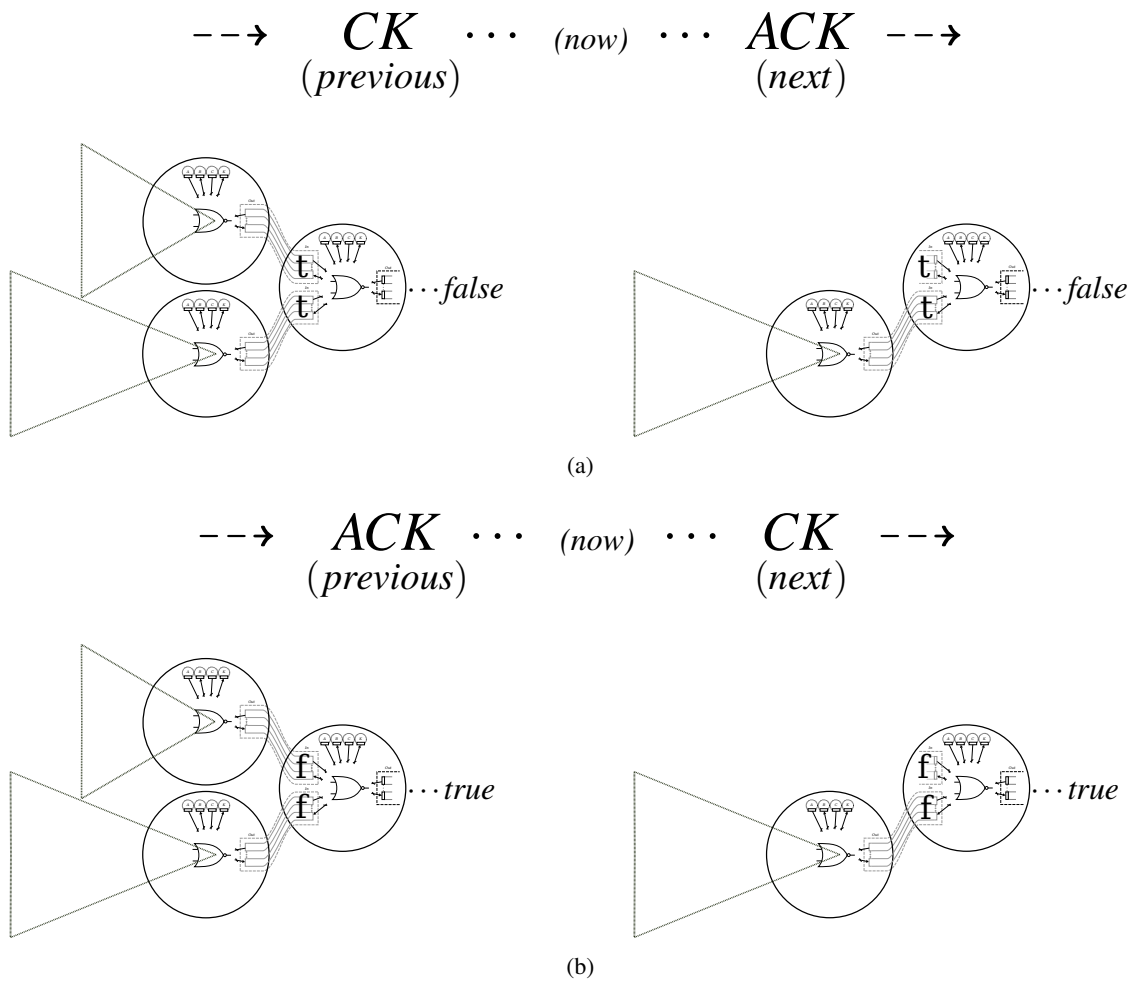


Figure 4.9: NOR-unit trees under the influence of an oscillating *A* signal. When the next signal group is *ACK* (4.9a), the previous signal group was CK, and therefore the previous output of the connected NOR subtrees is *true*. (NOR subtrees of arbitrary depth are represented by a NOR unit with an attached triangle.) Since the *ACK* signal group also sets an open *A In*-port to true, the root output after this group will be *false*. In (4.9b), the previous signal group was *ACK*, and therefore the previous output of the NOR subtrees is *false*. The next *CK* signal group does not change the default *false* value at an open *In*-port, and so the output of the root is *true*.

Proof. The output state of every individual NOR unit or those connected in binary tree structures will eventually oscillate out-of-phase with oscillating *A* or *B* signals. □

Blinking is an easy way to deterministically synchronize the output of every NOR unit in a CORAL environment in a distributed way. Though not proven above, the number of oscillations in the background signals for the above to apply needs to be at least the depth of the largest tree structure, since that depth defines the number of clock signals needed to advance all leaf-input calculation to the root of a NOR tree. We mostly ignore this complication in the proof above by not explicitly tracking the number of signal group oscillations in the recursive assumption.

Interestingly, and usefully, the same property above also applies to NOR meta-units. These structures perform the same NOR calculation but require extra time and signals to do so, proportional to their size. A tree structure made from meta-units such as the one shown in Figure 4.6 will oscillate out-of-phase with a background signal of triple the period:

$$ACK\ CK\ CK\ ACK\ CK\ CK\ ACK\ CK\ CK \ldots$$

A meta-meta-unit requires a period 3 times as long again. Not all the unit output states in the meta-structure oscillate, only the root units, though this is naturally defined as the output of the entire meta-unit in the next section. Structures composed of individual NOR units and those composed of meta-units of various scales respond differently to oscillating signals with longer periods, and this can be used to distinguish these structures using only background signaling.

4.3.2 Structure shielding

As the second property necessary for waterfall selection, individual NOR units have the inherent ability to ignore or "shield" the background signals *A* or *B* when connected at the corresponding *A* or *B* *In*-port and instead accept the previous output value of the *In*-port connected unit. Meta-units do not necessarily share this trait, since they are composed of units with both types of open *In*-ports (see Figure 4.5) and component units with open ports are always affected by those signals. However, if meta-units are connected in a particular way to one another and background signals encoded so as to cancel one another out as the calculation progresses, meta-units of any scale are also able to partially emulate the shielding property despite the presence of other, open ports.

Because of these many open *In*-ports, meta-units are potentially able to assemble and communicate with other units and meta-units in more ways than individual NOR units. However, for purposes here it is initially assumed that the only types of assemblies meta-units form are analogous to the assemblies individual NOR units form: binary trees. In the waterfall selection section below this assumption will be further justified, but defined here the only two relevant *In*-ports of a meta-unit are the upper *A* *In*-port of the topmost leaf node and the lower *B* *In*-port of the bottommost in Figure 4.5. These two *In*-ports, in combination with the *Out*-port of the root node of the NOR structure, correspond to the three ports of individual NOR units and are designated the *A* and *B* *In*-ports and *Out*-port of the *entire meta-unit*. The naming convention again reflects the behavior of the port discussed in Section 4.2, where a NOR unit or meta-unit connected to another at the *A* or *B* *In*-port ignores the background *A* or *B* signal and uses the connected unit's output value instead.

Figure 4.10 and 4.11 give an example of this shielding behavior in meta-units. If an *ACK ACK ACK* signal is received by a meta-unit with no meta-children, the output (at the root unit) will be *false*. Both the topmost leaf unit and the two bottom child units respond to the initial *A* signal, setting the output values of these nodes to *false* after *CK*. After the second *ACK* signal, the top child of the root unit has the value *true*, since it is connected at its *A In*-port, while the bottom child's output is now *false*. The third *A* signal cannot affect the output of the fully-connected root unit, so the final output of the structure is the NOR of the two child outputs, $(true \downarrow false) = false$. This calculation sequence is shown visually in Figure 4.10.

However, if the NOR meta-unit structure is connected at its *A In*-port (as in Figure 4.11) a different result is possible. When the input from the connected unit is *true* during the first set of *ACK* signals there is no difference in output, since in Figure 4.10 the *ACK* signals also set a *true* input to the open *A In*-port. If the input from the connected unit is *false*, though, the connection blocks the reception of the *A* signal at the topmost leaf unit (again see Figure 4.11). This results in the topmost leaf unit having a *true* output state. After the second *ACK* signal group, the bottom child of the root unit is set to *false* again because of its open *A In*-port, but the top child of the root node is now also *false*, since the top leaf unit previously output *true*. As before, the final *ACK* signals do not affect the root unit's output, which is again the NOR of the two child outputs, $false \downarrow false = true$. By sending interfering extra signals the meta-unit effectively ignores input not from an *In*-port. The same effect can be shown with *B* signals since the NOR meta-unit is symmetric.

Shielding the *In*-ports of a meta-unit by flooding with signals also works for larger meta-meta-units, no matter how many meta-levels are specified. There are an increasing number of open ports in these structures, but as above it is assumed only the topmost and bottommost child units from Figure 4.6 can be connected to other units along with the single *Out*-port at the root. Intuitively, this property is maintained in meta-meta-units for the same overall reason it was maintained in the meta-unit case - the bottom branch of individual NOR units or meta-units are always disabled to *false* by the continued *ACK* signals and this *false* input does not affect the result of the upper calculation. More information and a more detailed proof of shielding of meta-units is given in Appendix A, since the description is somewhat verbose.

Once the roots of particular structures have been enabled by blinking the meta-units at correct scales, input shielding allows us to select between structures using their connectivity. Waterfall selection can then be used to determine the positions at which the NOR units or meta-units of smaller scale will attach.

4.3.3 Waterfall selection

With the NOR-unit properties of blinking and shielding introduced above, it finally becomes possible to describe the full waterfall selection algorithm. To begin, every tree structure of NOR units has a unique root. From this root one can uniquely address every element in the NOR tree. For example, the root can be identified with the empty string ε and a simple naming rule applied where each child NOR unit in the tree is labeled by the parent name plus a *1* or *0* depending on the parent unit *In*-port to which it is attached. For the purposes here, the *A In*-Port connected units receive a *1* and the *B In*-Port units receive a 0. One can identify node in a tree given a name such as *11001*

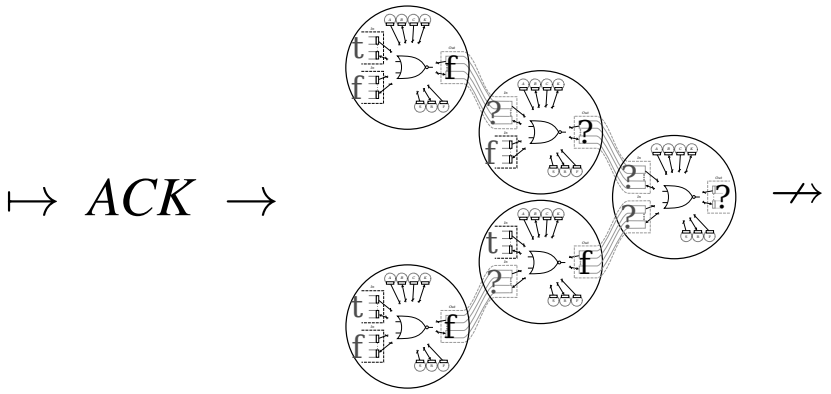


Figure 4.10: Values set from three *ACK* inputs propagating through a NOR meta-unit with free *In*-ports. After these signals are received, the root output of the meta-unit is *false*.

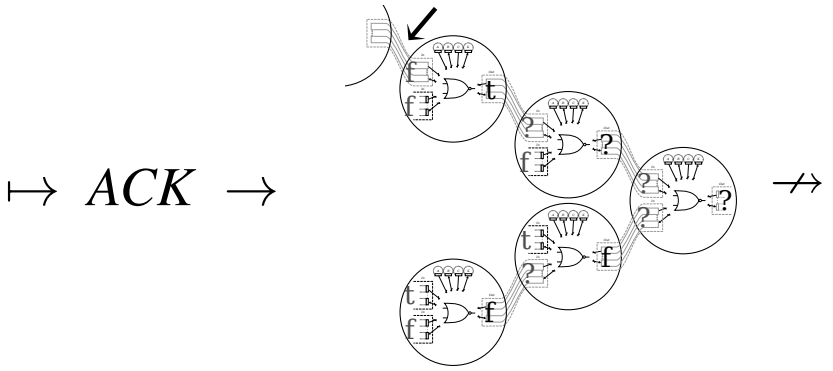


Figure 4.11: Values set from three *ACK* inputs propagating through a NOR meta-unit with a connected *A In*-port (indicated by the arrow). After these signals are received, the root output of the meta-unit is now *true*, as if the *A* signals had been ignored by the meta-unit's connected *A In*-port (though successive waves of calculation may be different, depending on further input from the connected structure shown partially here).

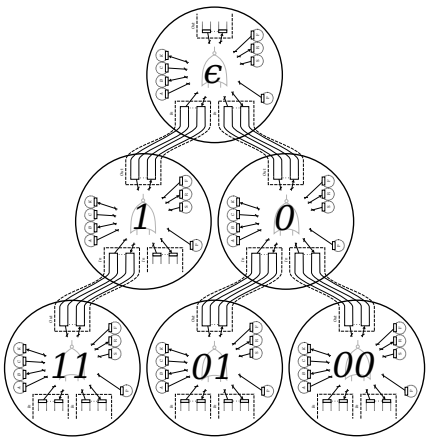


Figure 4.12: A NOR unit tree structure labeled by the waterfall selection naming rule.

by starting at the root unit and working downwards, choosing the next child based on the current character in the name string. One might think of a series of water-filled tubes flowing downward from the root, where at each node it is possible to switch a valve and change the direction of flow. Such a labeling is shown in Figure 4.12 with unit labels placed on top of the appropriate units.

Given a population of identical NOR tree structures made from NOR (meta-)units alongside individual, floating (meta-)units, one can first ensure that the only NOR units with *true* output state and without parent nodes are the root nodes of the NOR structures. This is done by blinking the units so that all have an output of *true*, and then sending a shielded input in order to *falsify* individual units but *not* units in structures. The roots of the meta-unit structures will then have a unique output value of *true*, which is used to start the waterfall selection process by priming these roots with *S* and *R* signals. The unit priming is then directed toward particular leaf unit ports at which new units are attached, using the label of the desired leaf unit as a "map" of how to get there.

Addressing a NOR structure in this top-down way implies a flow downward from the root, but the calculation of a NOR tree's logical expression flows upward from the leaves. There must be some path *downward* from the root unit, parallel to the upward path of logical operation, to specify where new units are attached. Progress on this parallel path is controlled via the previously-introduced *P* signal, such that a primed root unit will pass its priming to a child unit in response to this *P* signal. By relying again on the logical output state for direction (where *true* output sends the priming to the *A In*-port child, *false* to the *B In*-port), it is possible to transfer the priming of the currently primed unit to whichever child unit is chosen. As was proven above, every NOR unit in the simulation responds identically to an oscillating input, which gives a simple way of selecting the direction of priming flow at each unit. By following the label of the target node such that the units are blinked to *true* for a *1* character and false for an *0* character, followed by the *P* signal, eventually the priming will reach the target leaf node and the correct *In*-port of the growing structure will open. Figure 4.13 is an illustration of the process for a simple structure.

4.3.4 Full assembly algorithm

Figures 4.14 to 4.17 visually present the selection and growth of tree structures using the full waterfall selection algorithm. Each step requires only that blinking and shielding properties hold,

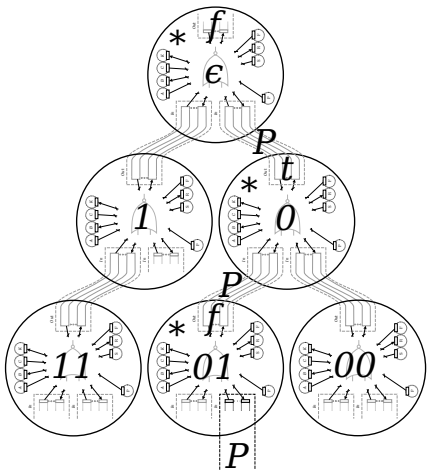
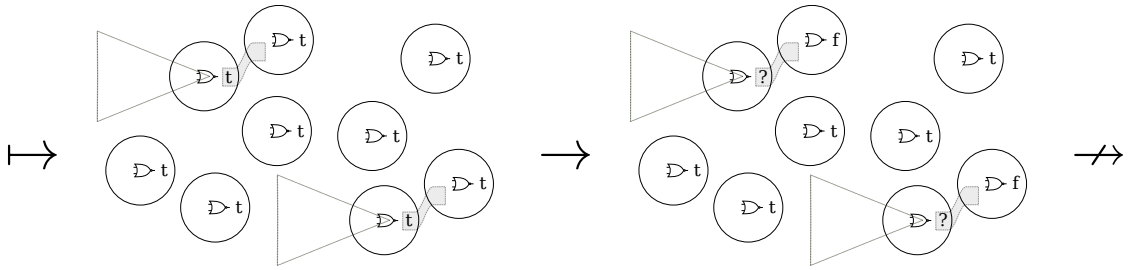


Figure 4.13: The NOR unit tree structure from Figure 4.12 demonstrating a priming path from root to final port on unit 01. Units are primed in sequence from top to bottom (starting with the root unit, primed by the *S* signal), and priming is indicated by a *. Blinking is used before each *P* signal to set the *true* and *false* output states corresponding to the next unit in the path, and the *P* signal then moves the priming to the next unit. When a leaf unit is reached the *P* signal opens the appropriate *In*-port of the unit for further assembly.

and these properties apply to both individual NOR units and meta-units. The waterfall selection algorithm, given these properties, is not only able to build any structure out of units or meta-units, but to do so over *indefinite scale*, allowing the control of arbitrarily structured trees of arbitrary size. Assembly of a meta-unit results in a structural quine, where the meta-unit responds in the same way but more slowly to the assembly commands. As can be seen from the recursive nature of the algorithm and verified by the provided source code, there is no algorithmic difference between building structures at small scale and building them at another, larger scale aside from the number of clock steps required to perform each of the program's child functions.

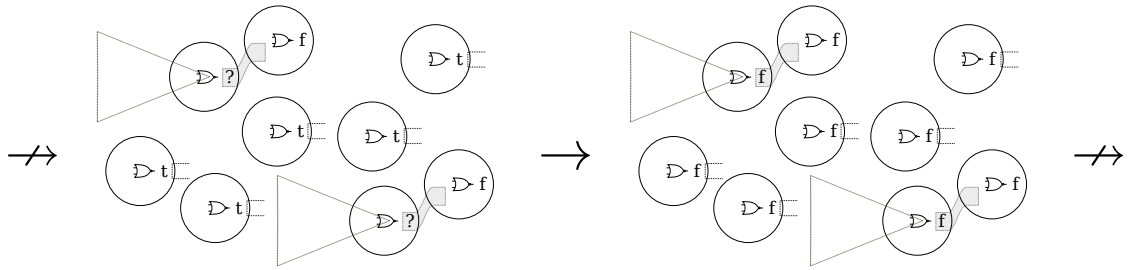
Algorithm 1 is a precise pseudocode description of waterfall assembly, using the functions `blink_structure`, `shield_input`, `break_symmetry`, and `select_waterfall` to encapsulate the respective operations. All functions implicitly write signal output to the `output_commands` variable, such that after a executing the `waterfall_assembly` function the signals can be read from this shared variable.

In the pseudocode presented, a `tree` specifies the structure to be built, while the `scale` indicates which scale of meta-units should be used as a basis for the structure. The five-unit meta construction is specified as the `META_UNIT_TREE` constant, which is built identically to any other structure but allows further assembly by preserving the necessary properties for meta-construction. See Section 4.2.1 for more details. The implementation given in the source code also shields against smaller left-over meta-units (which often exist due to stochastic partitioning of units) but the slightly modified logic required is not included to maintain the pseudocode algorithm's overall readability.



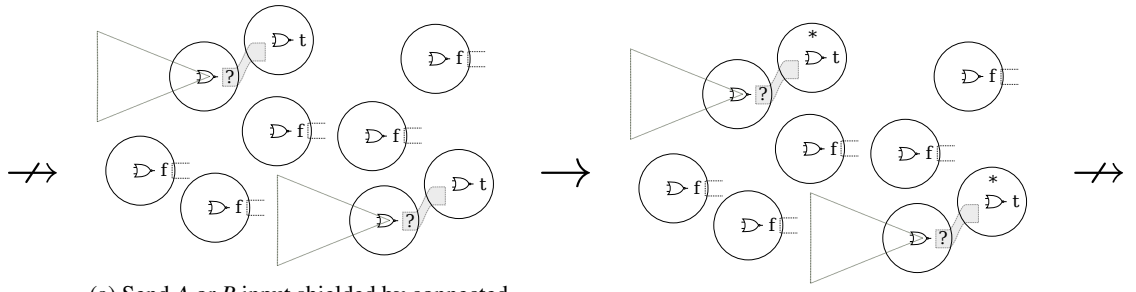
(a) Blink to *true.* (b) Advance with no *A* or *B* input.

Figure 4.14: Steps 1 and 2 of the waterfall selection algorithm. Originally, units are assumed to be in either single (meta-)unit structures or linked in copies of one particular tree structure. (In these next figures, "unit" is taken to mean (meta-)unit, and units are represented by circled NOR symbols to emphasize that they may be composite structures. This is a recursive assumption, and is maintained during the next steps of the waterfall selection algorithm.) Using the blinking mechanism described earlier, the output state of all units is first put into a state of *true* using oscillating input signals. Because all sub-trees of the tree structure are also subject to blinking behavior using oscillations of shorter duration, all sub-trees will also have a root output state of *true*, as can be seen in (4.14a). Next, a potentially repeated empty signal is sent, with duration depending on unit size. In the case of single units, this empty signal results in a *true* output value, as can be seen in (4.14b). In tree structures, however, the root unit is passed the sub-trees' previous *true* value, resulting in a *false* output which is used in the next step to discriminate between single and structured units.



(a) Open output of individual units with *F.* (b) Blink to *false.*

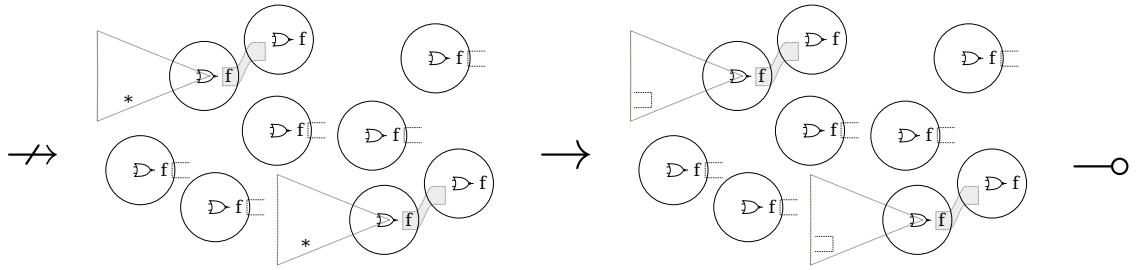
Figure 4.15: Steps 3 and 4 of the waterfall selection algorithm. Continuing from Figure 4.14, units and trees of units are now distinguished by root output state. Per the operation of units described earlier in Section 5.4, using the *S* and *F* signals allows us to open the *Out*-ports of the individual units only (4.15a). In (4.15b), blinking is again used to normalize the output of all units, trees, and sub-trees to *false*, in preparation for another discrimination of trees from individual units.



(a) Send *A* or *B* input shielded by connected port.

(b) Prime with *S* signal, direction *R*.

Figure 4.16: Steps 5 and 6 of the waterfall selection algorithm. Again continuing from Figure 4.15, all units and trees initially have *false* output. The shielding property of units and meta-units now allows us to discriminate trees from individual units once again, by sending an *A*, *B*, or either signal depending on whether the root unit of the tree structure is connected at *A*-, *B*-, or both ports. This signal is shielded by the connected unit if one exists, resulting in tree structures with an output of *true* and individual units with an output of *false* (4.16a). Again using the signals from Section 5.4, the root unit is primed (4.16b) and ready to select an *In*-port to open, as was seen in more detail in Figure 4.13.



(a) Blink and use *P* to move priming.

(b) Priming reaches leaf unit.

Figure 4.17: Steps 7 and 8 of the waterfall selection algorithm. In Figure 4.16, the root units of tree-structures are primed. The *In*-port selection process is now continued by progressively blinking the tree structure and moving the primed state backwards through the tree structure (4.17a), which is again shown in more detail by Figure 4.13. Once our priming state reaches the leaf unit that we have chosen for the attachment of a new individual unit (4.17b), the leaf unit *In*-port is opened, which allows the assembly of a new unit to the tree structure. Once left-over individual units are reset, this results in the original state seen in (4.14a) above, but with a larger tree structure. By successive application of these steps, any tree structure can be "grown" from a collection of individual units and a tree seed generated via symmetry breaking. The full assembly algorithm in pseudocode is presented in Section 4.3.4.

Algorithm 1 Waterfall selection assembly algorithm in Python pseudocode.

```
# Constants
META_UNIT_TREE = ...; ASSEMBLY_WAIT = ...

output_commands = [] # Stored commands

def waterfall_assembly(tree, scale):

    # Assemble meta-units to correct scale
    for s in range(0, scale):
        assemble(META_UNIT_TREE, scale)

    # Assemble tree of meta-units
    fringe = [(tree, "", 1)]; max_depth = 0
    shield_dir = ("A" if tree.children[0] else "B")
    while len(fringe) > 0:


        unit, address, depth = fringe.pop(); if not unit: continue
        max_depth = (max_depth if max_depth > depth else depth)

        # Assemble the current unit and add children to the fringe
        assembly_step(address, scale, max_depth, shield_dir)
        fringe += [(node.children[0], address + "1", depth + 1),
                   (node.children[1], address + "0", depth + 1)]

# Add a unit to the growing tree at a particular address
def assembly_step(address, scale, max_depth, shield_dir):

    # Select individual units
    if address != "":
        blink_structure("A", "A", max_depth - 1, scale)
        empty_input(1, scale)
    else: break_symmetry(scale)
    send_signal("S"); send_signal("F")

    # Select structure trees
    if address != "":
        blink_structure("A", None, max_depth - 1, scale)
        shield_input(shield_dir, scale)
    else: empty_input(1, scale) # Select remaining units
    send_signal("S"); send_signal("R")
    select_waterfall(address, scale)
    send_signal(ASSEMBLY_WAIT)

# Send empty input for a particular scale
def empty_input(length, scale): ...
# Send a general signal input
def send_signal(signal): ...

# Core waterfall selection functions
def blink_structure(a_or_b, last_signal, depth, scale): ...
def shield_input(a_or_b, scale): ...
def select_waterfall(address, scale): ...
def break_symmetry(scale): ...
```

4.4 Implementation in 36 bits

The final NOR unit with all necessary signal transitions and port transitions assigned is shown as Figure 4.18 in diagrammatic form and with all internal logic as Figure 4.19. There are 36 places and 41 transitions in the full C/E net, hence the title of the section. Actually several places serve half duty, acting as one part of an on-off toggle (since the C/E nets we use lack inhibitory connections), so markings represented as bit strings might actually be a few bits shorter. Though the image at first may seem complex, the NOR unit implementation behaves exactly as described above except for two final signals (described below) needed for simulated stochastic symmetry breaking in non-infinite numbers of units. Much of the repeated structure also arises from the fact that, in the CORAL environment, a unit must explicitly track when it becomes linked to other units if this changes the unit behavior. There is no default environment-controlled state available to the unit in which this information is stored. A second complicating factor is that NOR units must ensure that some signal transitions fire only a single time per signal when it is a requirement of the control algorithms defined above. In real environments with real devices (and in many other simulated examples), often these two requirements are trivial because they are implicit properties of the environment. It seems there is quite a bit of interfacing which often goes unremarked in these more complex interactions and is exposed computationally here by the more minimal CORAL framework.

To break the initial finite set of identical units or meta-units into two groups, an extra *Y* signal is used. All units and meta-units begin in the same state, and so these must be partitioned using some stochastic method (since addressing individual units is not possible) into a set of "seed" and "stock" units of correct proportions. Structures and meta-units grow downward from the seed units, using waterfall selection to determine where the next stock unit is added. Stochasticity in any of the transition firing timings could also have been used to partition the initial set, though the dedicated signal provides a slightly more straightforward way of selecting fractional sets of units. The assembler described in Chapter 5 uses transition timings instead. An additional *Z* reset signal is also needed to undo assembly steps if, due to an insufficient number of stock units, open *In*-ports of growing structures are unable to be paired. The stochastic partition, by definition, does not always work perfectly, especially for smaller numbers of units.

Two directional transitions per port are used in the implementation of Figure 4.19, one to pass the NOR calculation tokens forward and one to send a priming token backward, and as mentioned earlier two places per port are used to track whether the port has been connected or not. The C/E diagram data used to generate Figure 4.19 and execute it in the CORAL simulator is available online at:

- https://coralassembly.wordpress.com/

The software used for design and layout of the many C/E net graphs, both for display in this thesis and in simulation, is the free yEd graph editor, written in Java and available for all platforms at:

- http://www.yworks.com

Graphs designed in yEd can then be translated directly into C/E nets, such that the implemented NOR unit displayed below can be imported as-is into a simulation.

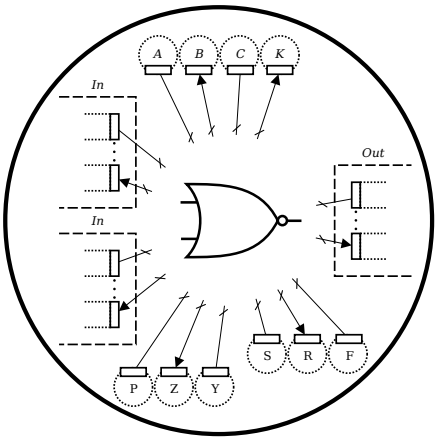


Figure 4.18: Diagram of a NOR assembler with all signals and ports needed for waterfall selection (P), breaking symmetry (Y), and resetting incomplete assembly (Z).

4.4.1 Simulated assembly of meta-units

Using this implementation, we can show scalable assembly working to create meta-units, meta-meta-units, and so on in a simulated CORAL environment. As before, we must set our assembly rate α and transition firing rate τ: here $\alpha = 1$ assembly operation per time step and the max time steps per transition firing is $\tau = 20$. The value for τ is high here so that there is a gradient in transition firings when breaking symmetry, however this also means that the simulation must often wait several time steps between setting signals so that transitions will fire. In the skeleton signal strings shown in the captions of simulation figures the wait times are omitted for clarity, but they are generated by the sample code provided. Figures 4.20 to 4.25 illustrate the process of creating structure and scalable assemblers for a sample system with 512 NOR units.

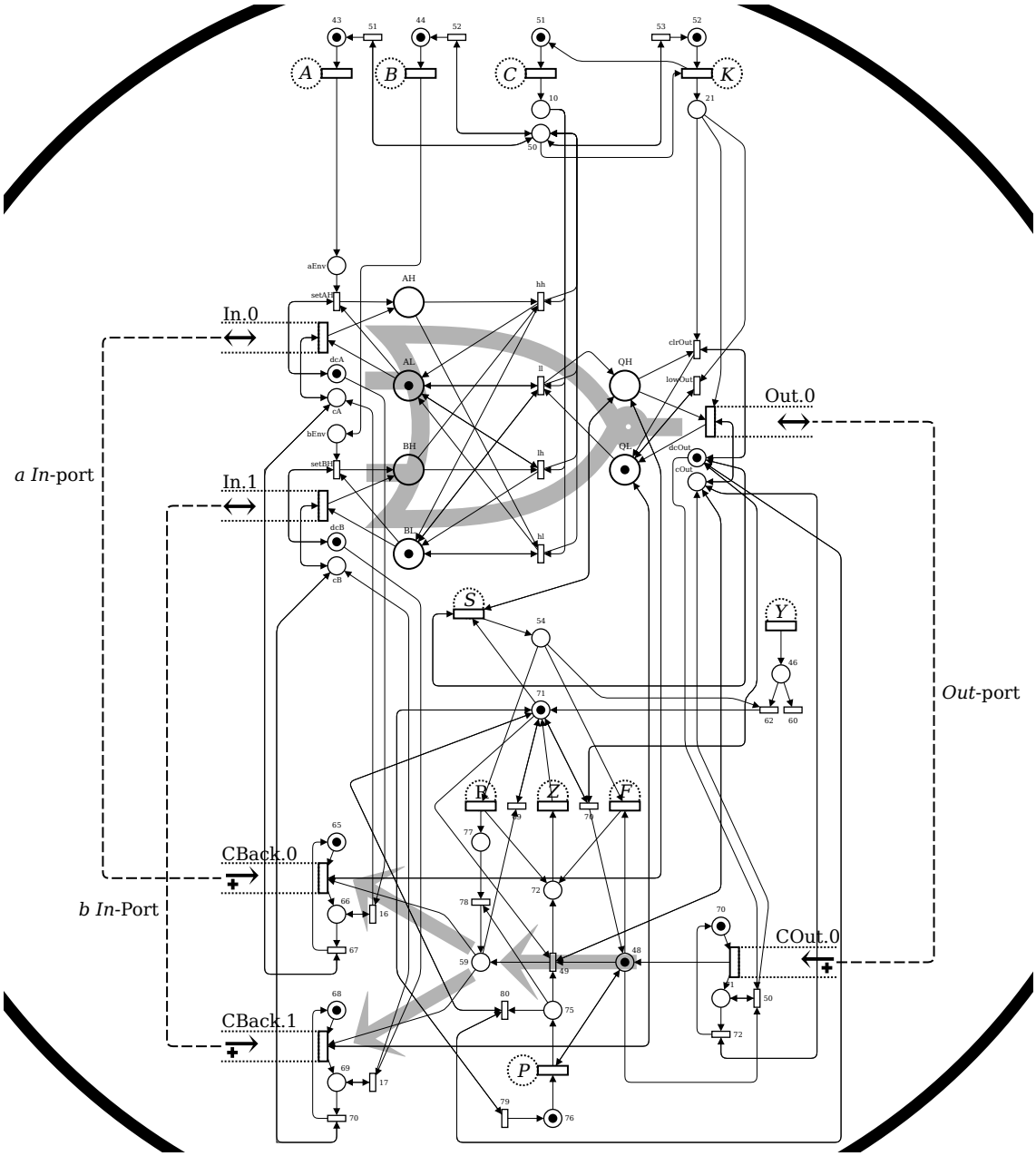
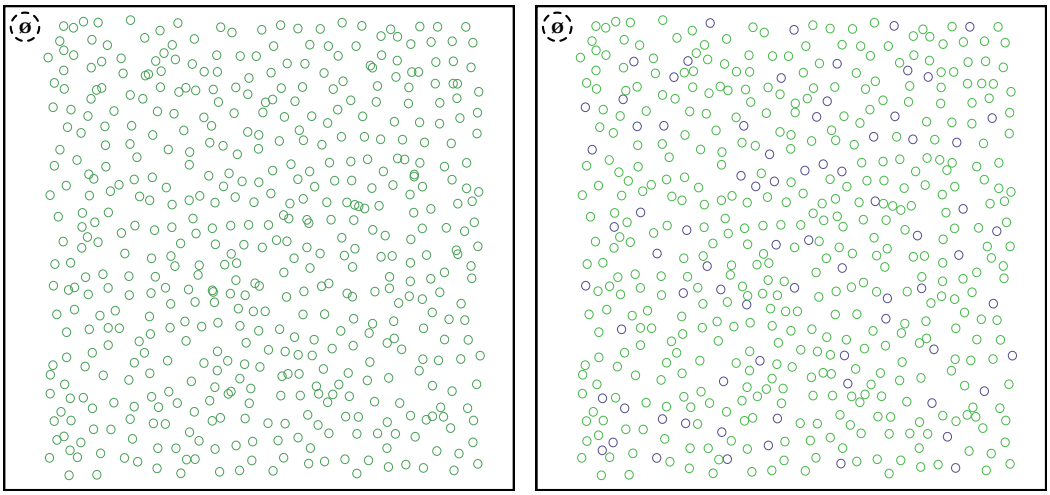


Figure 4.19: An implementation of the final NOR assembler diagrammed in Figure 4.18, zoomed in to highlight the controller. Certain signals and ports are placed so that the edges and logic are as clear as possible. Generally, the top section of the C/E net handles NOR inputs from signals and ports (indicated by the NOR symbol), producing a NOR output at the Out.0 transition after a *K* signal. The lower half of the C/E net manages assembly and the flow of waterfall selection in the opposite direction (indicated by the branching arrows). The output state of the unit, discussed in abstract in the sections above, is contained in the two large places QH and QL and is linked to the waterfall assembly logic by the S and CBack transitions. Much of the duplicated logic around each port transition tracks whether or not the port has been connected, and much of the duplicated logic around signal transitions ensures the environment only fires the transition once. In particular environments, this extra processing may not be necessary.



(a) (b)

Figure 4.20: A CORAL simulation containing 512 instances of the NOR unit implementation, initially in identical state (4.20a). Unit controllers are not drawn due to the number of units pictured. Stock units are selected by blinking all units to *true* (for structures of depth 1, blinking is just a single clock signal), disabling selection on some of them, then enabling forward connections: $(C)_{blink}, (Y, : wait)_{sym\_br}, (S,F)_{open}, K$. (Portions of the signals corresponding to each operation are labeled with subscripts.) Seed units are the remainder and selected for reverse connections: $(C)_{select}, (S,R)_{open}, K$. The resulting seed and stock units are shown in different colors and brightness as they are in different states (4.20b).

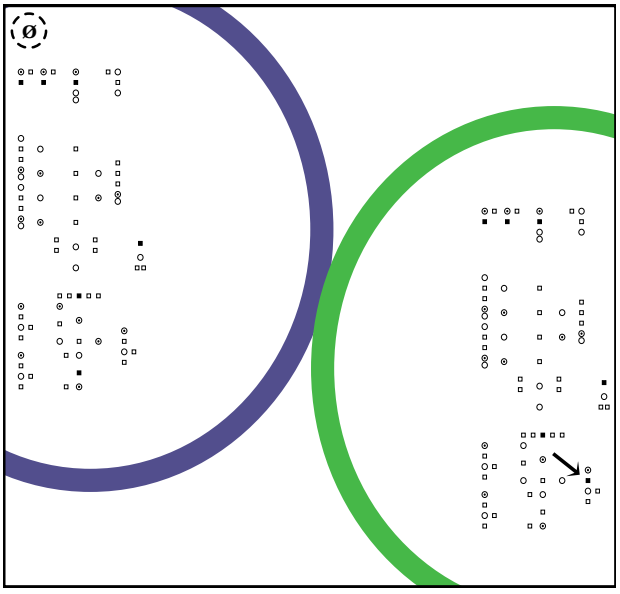
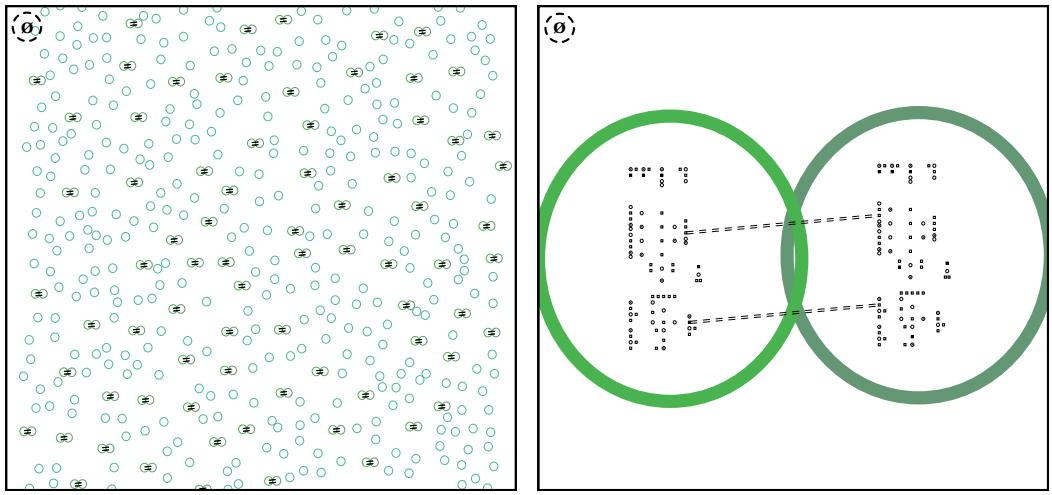
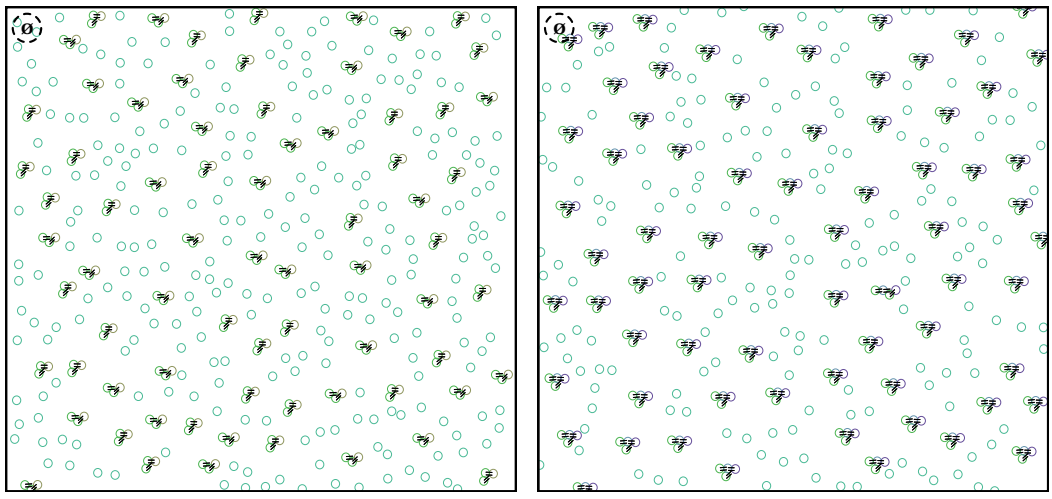


Figure 4.21: Close-up comparing the controller of seed and stock units from Figure 4.20b. The controller is the one pictured in Figure 4.19, however edges are not drawn for clarity. The seed and stock units are in different states, indicated by color of the outer ring, and also by the different markings of each unit. The arrow indicates the *Out*-port link transition, which is enabled for all stock units but not for units of the growing structure.



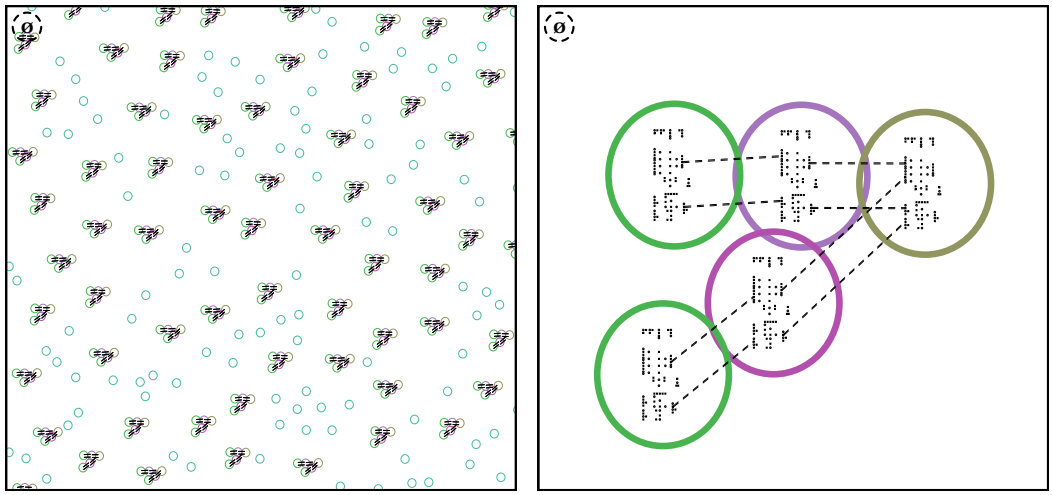
(a) (b)

Figure 4.22: After selecting the *A In*-port of the seed units from Figure 4.20b (by setting the output state to *true*) using the one-unit case of waterfall selection, units assemble in the environment: $(C,P)_{wfall},(:wait,Z)_{assm},K$ (4.22a). A closeup of the linked controllers and synchronized transitions of the assembled pairs is shown as (4.22b), drawn again without C/E net edges.



(a) (b)

Figure 4.23: Two further stock units are added to the growing structure using the same blinking and waterfall selection technique. From Figure 4.22a, individual stock units are selected by blinking all units to *true*, then sending an empty input of *false* while connected units are disabled by receiving *true* from their children: $(C,K)_{blink},(C)_{inp},(S,F)_{open},K$. Seed structures are then selected in the opposite way, by blinking all units of current scale to *false*, then sending a shielded $a = true$ input to the connected port connected units will ignore: $(A,C,K)_{blink},(A,C)_{shield},(S,R)_{open},K$ and afterwards using waterfall selection $(A,C,K,C,K)_{blink},(A,C,P)_{wfall},(:wait,Z)_{assm},K$. Similar commands are used for the third unit.



(a) (b)

Figure 4.24: Many instances of the final assembled meta-unit from Figure 4.5 are completed (starting from the environment of Figure 4.23b) and shown in (4.24a), again using the same the same blinking and waterfall selection technique. A close-up of the meta-unit is shown in (4.24b). This meta-unit can now be used as a base for future structures, as is shown in the next Figure 4.25.

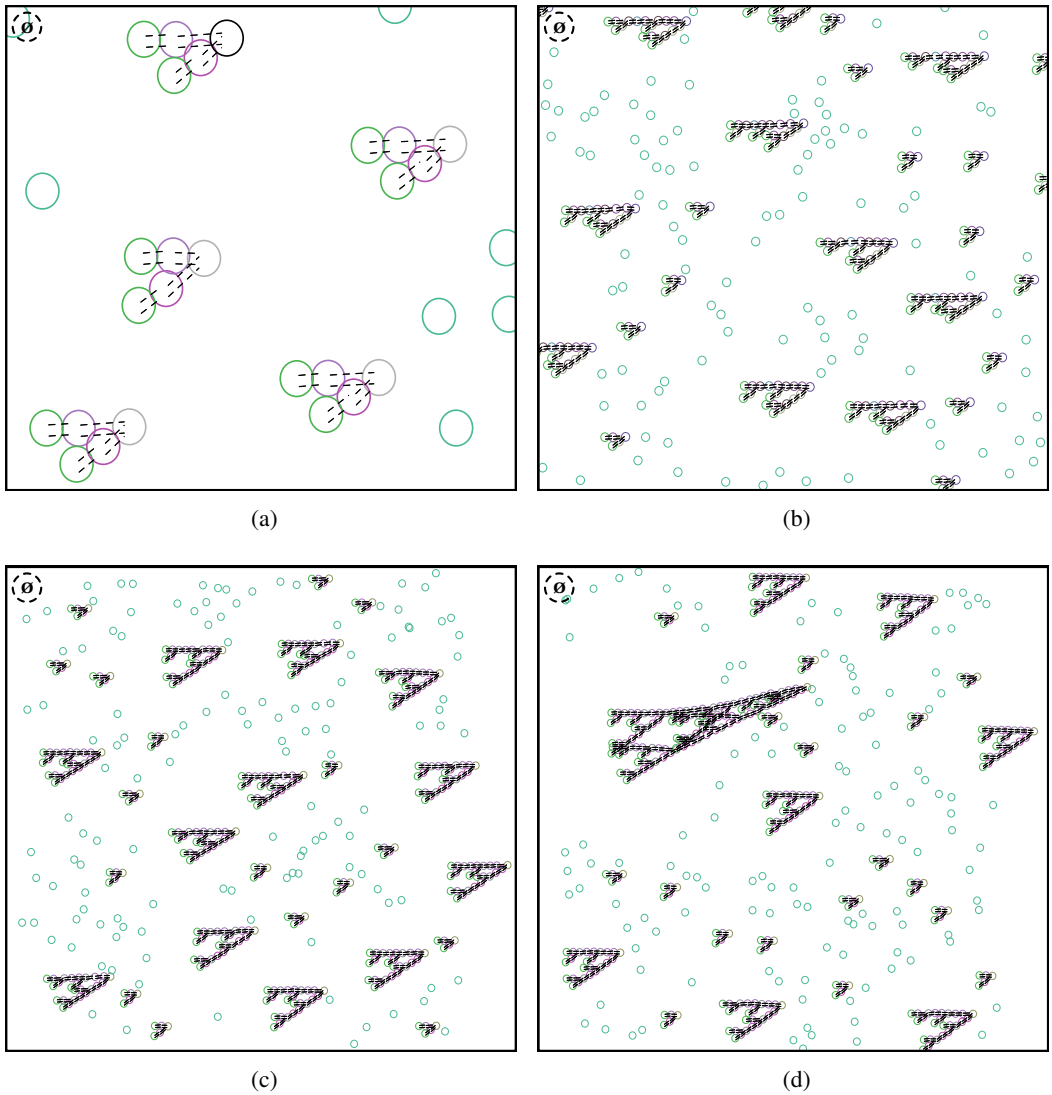


Figure 4.25: Using the same steps as were used to create meta-units (or any other structure) from units, meta-meta-units can be created from meta-units (as in Figure 4.6). The simulation continues uninterrupted from Figure 4.24 and the initial symmetry breaking step is depicted in (4.25a) (note the colors of the root NOR units). Subfigures (4.25b) and (4.25c) are built using higher-order blinking, shielding, and waterfall selection, and (4.25c) illustrates 12 created meta-meta-units. A final third-level meta-meta-meta-unit was created using the same process from meta-meta-units in 4.25d (parts of the structure lie atop one other in 3D).

4.5 Emulating arbitrary formulae

With the demonstrated ability to form arbitrary structures at any scale, it is now possible to show how formulae with multiple variables can be emulated using particular types of these structures. In general, any logical formula can be converted into a form using only nested NOR operations. For example, as was shown in Equation 4.2, $a \vee b = (a \downarrow b) \downarrow (a \downarrow b)$. A NOR unit structure can emulate this and any other nested NOR formula. The construction is simple and natural: starting from the outermost NOR operation, attach a child NOR unit for each child NOR operation to the

corresponding *In*-port, then do the same recursively for each child unit. The construction for $a \vee b$ is shown as Figure 4.26. If the input values for the corresponding NOR units are set to variable values, after a number of clock steps equal to the NOR structure depth the root output of the NOR structure contains the result of the formula.

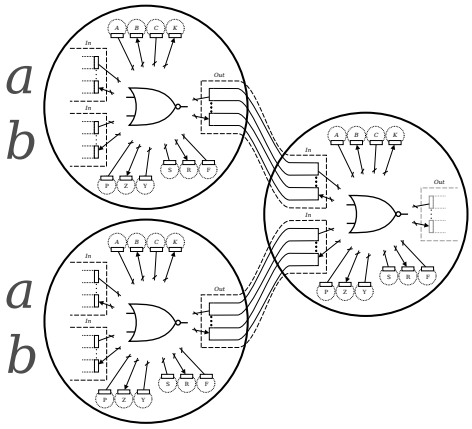


Figure 4.26: NOR unit structure for $(a \downarrow b) \downarrow (a \downarrow b)$.

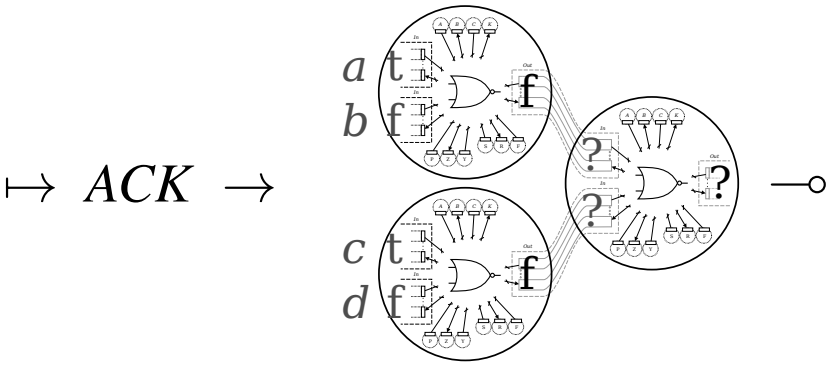
A problem arises when logical formulae contain an unlimited number of variables, not just a and b. For example, consider a NOR formula similar to the one above: $(a \downarrow b) \downarrow (c \downarrow d)$. This is a valid expression and using the natural construction is transformed into the same NOR unit structure as in Figure 4.26. This time, however, the four *In*-ports of the leaf NOR units take as input the values of variables a, b, c, and d. To set these variables such that the output at the root unit takes them all into account, the values of all four must be set prior to the same clock step. Since there are only two signals, A and B, available to set four values, a and c will always be equal using the Figure 4.26 construction, as will b and d. Figure 4.27 illustrates this collision in variable value assignment. Similar collisions are possible even using two variables when the child trees are of different depths, though generally it is possible to swap *In*-ports in the NOR assembly (since NOR is commutative) so that such collisions are eliminated.

To avoid these variable value collisions, an additional "antennae" construction is defined here which allows each variable to be specified at a different offset in clock time. Larger antennae receive earlier A, B signals and feed this data at the appropriate clock step into ports of the core logical construction. In the conflicted translation of the example expression $(a \downarrow b) \downarrow (c \downarrow d)$, each variable value is input simultaneously at the depth-2 leaf nodes of the structure. If these variables are labeled with this depth and signal which they are input, the result is as follows:

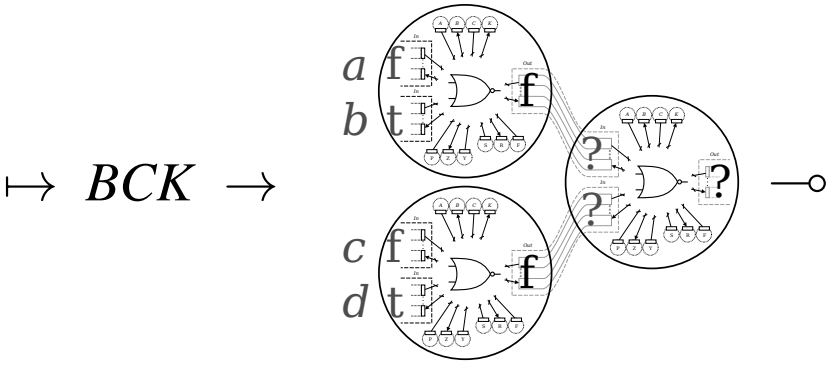
$$(a_{2,A} \downarrow b_{2,B}) \downarrow (c_{2,A} \downarrow d_{2,B})$$

All variables with a shared depth and signal cannot be specified independently. Ideally, these signals and depths would each be unique, allowing the specification of any set of values in a logical formula. To simplify the problem somewhat, one can require that only the depths be unique. As will be explained below, using the same signal to specify each variable helps resolve conflicts in the antennae themselves.

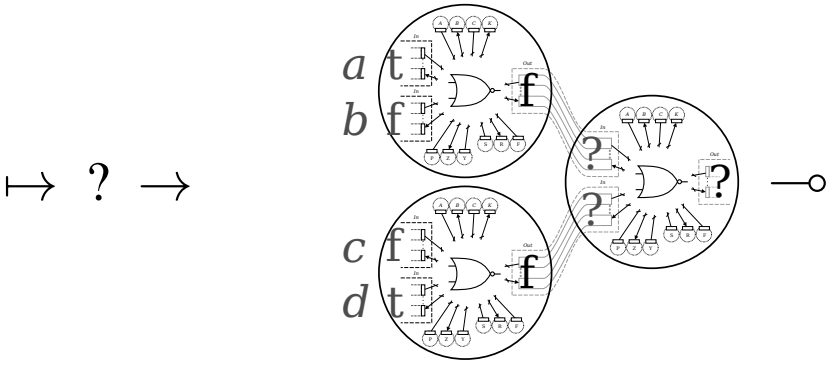
The antennae construction allows the depth of each variable to be varied without altering the output of the logical expression. It is based on a simple logical identity hinted at earlier:



(a) $a = c = true, b = d = false$



(b) $a = c = false, b = d = true$



(c) $a = d = true, b = c = false$

Figure 4.27: Various background signals which assign the values of variables for the NOR unit emulation of $(a \downarrow b) \downarrow (c \downarrow d)$. In (4.27a) and (4.27b), correct values for each input can be assigned using background signals. Not all combinations of variable values can be specified, though, as shown in (4.27c), because in this case the values require opposite inputs be sent to signal transitions of different units in a single clock step. Selective activation of units this way is impossible in the CORAL model.

$$x \downarrow false = \neg x$$

and therefore:

$$(x \downarrow false) \downarrow false = \neg(x \downarrow false) = \neg(\neg x) = x$$

Using this identity, it is possible to nest a variable arbitrarily deeply inside a logical formula without changing the value of the output. A new symbol is assigned for a nesting of this type, evocative of an antennae itself:

$$\begin{aligned} \rightsquigarrow (x)^0 &= x \\ \rightsquigarrow (x)^n &= (\rightsquigarrow (x)^{n-1} \downarrow false) \downarrow false \end{aligned}$$

It is also easy to show recursively that:

$$\rightsquigarrow (x)^n = x$$

Therefore, we may rewrite the example expression $(a \downarrow b) \downarrow (c \downarrow d)$ as:

$$(\rightsquigarrow (a)^0 \downarrow \rightsquigarrow (b)^1) \downarrow (\rightsquigarrow (c)^2 \downarrow \rightsquigarrow (d)^3)$$

When constructing this formula using NOR units, extra, even-length chains of singly-connected units are connected to the appropriate *In*-ports of the core logical structure. Each $(x \downarrow false) \downarrow false$ formula requires two NOR units to implement, connected like so:

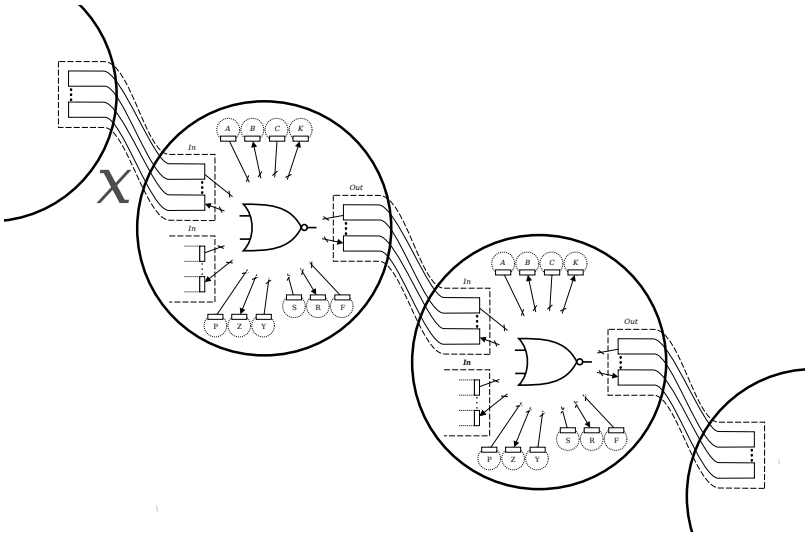


Figure 4.28: NOR units which can emulate the $(x \downarrow false) \downarrow false$ function as part of a longer antennae. There is an equivalent construction using *B In*-ports, by the commutivity of NOR.

Using this NOR structure, an antennae $\rightsquigarrow (x)^n$ of length n requires that the variable x be specified at a NOR structure depth increased by $2 \times n$. Since antennae of any length can be substituted for a variable, one simply needs to select antennae of unique lengths in order to rewrite any formula

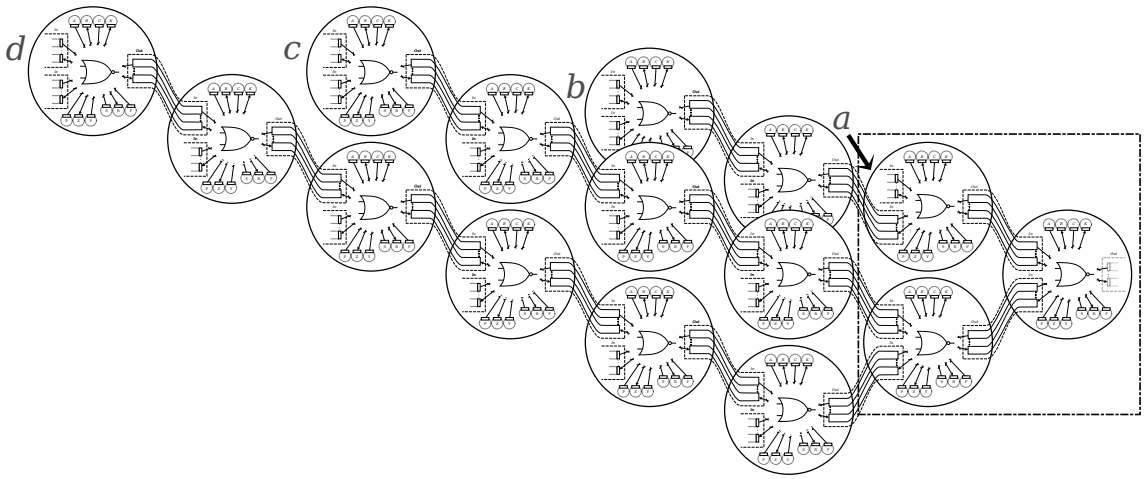


Figure 4.29: The NOR structure emulating $(a \downarrow b) \downarrow (c \downarrow d)$ (in dashed box) with attached antennae units. The variables a, b, c, and d are now input at unique depths, allowing each value to be set independently.

so that variables do not conflict when specified. Applying this to the rewritten formula, labeling each variable with depth gives:

$$((\rightsquigarrow (a_2)^0 \downarrow \rightsquigarrow (b_4)^1) \downarrow (\rightsquigarrow (c_6)^2 \downarrow \rightsquigarrow (d_8)^3))$$

The particular signal specifying each of the variable values has been omitted in the above formula. In Figure 4.28, it is assumed that the x input variable is specified through the topmost *A In*-port of the leaf unit, and that the other input to the unit will be false. If a *B* background signal is set in the environment while the x value is propagating, however, the x value will be ignored and output passed to the next unit forced to *false*. If there are multiple antennae of different lengths, each passing a previously set variable value to the core logical NOR structure, setting a new variable value using *B* signals will interfere with all previous values. For this reason, we assume that all variable values are set using an *A* signal, with the depth of the antenna construction controlling which variable is actually being set at a particular clock step. So long as these depths are unique (and they can be, since antennae can be constructed of arbitrary lengths), using only a single signal to set variable values avoids all conflicts and allows the value of any number of variables to be specified in any formula. Figure 4.29 shows the final multi-variable NOR structure with antennae attached. Structures made from meta-units can be built in the same way with corresponding meta-unit antennae since the identities used for logical equivalence still hold.

Using antennae built from NOR units, every formula in propositional logic can be emulated by a NOR unit structure. The value of each variable is specified at a particular depth, and the output of the structure can be read off the output state of the root unit after a number of clock steps equal to the depth of the unit tree. For the unit in Figure 4.29 it is now possible to calculate the result when $a = d = true$, $b = c = false$ by setting the following signals sequentially in the environment:

$$\begin{aligned} &ACK \text{ (for } d = true) \; CK \\ &CK \text{ (for } c = false) \; CK \\ &CK \text{ (for } b = false) \; CK \\ &ACK \text{ (for } a = true) \; CK \\ &CK \text{ (final result)} \end{aligned}$$

4.6 Remarks

NOR units have been shown above to be fully general assemblers of logic expressions, due to their properties of scalable assembly and the antennae construction. It is possible to build any tree structure of NOR units using the implementation shown in Figure 4.19 and the background signals generated from Algorithm 1. Since these trees of NOR units correspond exactly with nested NOR expressions, and from nested NOR expressions one can build all other logical expressions, one can build a NOR device to calculate anything described in propositional logic. NOR meta-units also have at least the same expressiveness, given their ability to assemble into arbitrary binary trees linked by extreme *A* and *B In*-ports. Though the CORAL model used to simulate NOR units is very abstract, it relies only on complementary pairing and the sensing of shared background signals. Both of these properties have been demonstrated by a variety of natural and artificial assembling systems to varying degrees, and so it is not unreasonable to assume that NOR assemblers could actually be built from electronic, mechanical, or perhaps even chemical parts.

Though Chapter 8 discusses the implications in more detail, the results demonstrated here present NOR units as a prototype of realizable, artificial assembling units capable of building arbitrary levels of organization and arbitrary complexity at all these levels. While not shown to be a minimal implementation, it is worth remembering that Von Neumann's original universal constructors were rather unwieldy, and only successively refined over time. Simply having a deterministic, easily accessible model which generates these scalable structures is a major step for further advancements. As was the author's experience, hypotheses for minimal future directions become much easier to form and evaluate when there is a model already creating the behavior you wish to optimize.

With that caveat, the NOR unit implementation appears more complex at first glance largely because it contains an explicit, computational description of properties other environments might take for granted. Port disablement after connection and single responses to input signals are two of these. Even so, as mentioned above, the amount of memory required to store the states of such a unit (36 bits) is tiny compared to the vast megastructures the units can deterministically *and* dynamically form (dynamically, because certain megastructures can always form other, bigger structures). In effect, almost all the complexity in the target structure has been offloaded indirectly into background signals - and there are probably further refinements making units even simpler. Even at this stage, however, NOR units, by adding the ability to scale assembly over orders of magnitude, allow the possibility of generating from microscopic parts macroscopic devices of arbitrary precision.

Beyond a proof-of-concept for assembly control over very large parts, the choice of propositional logic was made not only for abstract reasons but because the output of this logic can also be attached to other processes. The CORAL model is solely concerned with questions of assembly - there are no 'physics' or even 'locations' to allow structural actions. These actions have the potential to be highly interesting, however. For example, unit-compressible assembling modules with only the ability to change horizontal or vertical dimension (Rus & Vona, 1999; Suh et al., 2002; Ishiguro et al., 2006) have been shown to be highly capable assembling devices, alternately so are devices which can simply modify angles between two parts (Hamlin & Sanderson, 1997; Ünsal & Khosla, 2000; Tuci et al., 2005; Zykov et al., 2007; Detweiler et al., 2007; Kirby et al., 2007). Such an ability could be added to a real-world implementation of a NOR unit, allowing structures of these units to manipulate their shape, linked to their logical output state. If, for example, compression was directed by the current logical output of the NOR unit, NOR structures can be constructed which have dynamic shape manipulation directed by the value of logical expressions. Other types of actions are also possible. Importantly, these actions would be controllable in similar ways for *structures of arbitrary size*. Only preliminary work has been done to date demonstrating this idea, however. Even more capable designs are possible if structures at multiple scales can be merged, *building a brain alongside its body*. Chapter 8 discusses this idea further and presents a hypothetical unit-compressible action linked to a NOR assembler.

While logical output is interesting, it also has limited expressiveness for doing these sorts of tasks. By nature formulae are static, and the behavior of assembled structures ideally highly dynamic. Also, binary tree structures are the only types of objects which have been built using the NOR unit approach, though this is not only because binary trees correspond to logical syntax but also because new difficulties arise once cyclic graph structures can be created. A different basis is needed for a more complex assembler, one which builds structures that instead approach the capability of arbitrary Turing machines. The next chapter presents that new basis as C/E nets themselves, with the next assembler able to build an emulation of itself or any C/E net at larger scale directly.

Chapter 5

Graph Assembly and Computation using Petri Assemblers

The NOR units of Chapter 4, while capable of building arbitrary meta-structures, only do so in the form of multi-scale binary trees. This is sufficient for Boolean logical expressions, which are represented as trees of sub-expressions. It would be interesting, however, if the same type of distributed assembly was capable of producing structures of arbitrary topology. Perhaps even more interesting would be if these structures had arbitrarily powerful potential for computation. In this chapter, a second CORAL assembler design is presented with both of these properties, directly demonstrating a CORAL assembling environment able to emulate any computer. Previous assembling artificial chemistries have built or emulated interesting computing devices, but the systems used have been either highly abstract (ignoring basic physical properties such as conservation of matter, e.g. (McCaskill, 1988; Fontana, 1992; Nagpal et al., 2003; Crutchfield & Görnerup, 2006; Salzberg, 2007)) and/or preprogrammed through some initial memory input. The assembly described here assumes only a mixture of basic, undifferentiated computational "amino acids," with the final result depending more on perturbations than the units themselves. The system is probably most similar to the more recent work of (Danos & Laneve, 2004; Klavins et al., 2006b) in which the operation of many distributed parts can be synthesized into and derived from high-level formalisms. This begs the question: is there a minimal useful part in such systems? One simple but powerful computational prototype is demonstrated here, from which any other computing device can be bootstrapped.

This assembling unit, the C/E unit, emulates base C/E net operations. These operations, as discussed in Chapter 3, are the core of every atomic unit in the CORAL model. A multi-level symmetry is created where assembling C/E units and signal information can form any larger, meta-Petri net - *including the atomic C/E controller itself.* This brings into sharp focus the novel aspects of a dynamic approach to assembly, since the C/E units emulate only a fraction of themselves and cannot be pre- or re-programmed with more than a tiny part of the whole pattern of their structure.

The chief challenges discussed in this chapter relate to deterministic graph assembly, introduced briefly at the end of Chapter 4, and the design of a minimal C/E net component. Additional complications arise from the potentially PSPACE-complete nature of subassemblies when build-

ing computing devices more powerful than boolean logic. A distributed state-shifting mechanism is introduced for this reason, based on linear-feedback shift registers (LFSRs), which allows general state manipulation of arbitrary numbers of units using only a very small number of signals. Because of this orthogonal addition, the C/E unit assembly does not yet scale recursively even with arbitrary meta-net emulation, though there is very probably scope for such a recursive assembly process to be created in the future.

5.1 Overall approach

The following chapter contains much technical detail, and so it is helpful to first sketch the overall direction and problems addressed while undertaking this line of research. In general, the goal is similar to that described in the previous chapter - the design of an assembling unit with particular compositional properties. This leads to certain similar problems which must be addressed:

- the creation of a syntactic and structural abstraction which defines equivalence between a collection of units and individual C/E unit functionality, similar to the way the logical identities of NOR units lead to a meta-unit abstraction in Section 4.2.1. The corresponding construction for C/E units is called *localization*.
- the design of a fully distributed assembly algorithm capable of linking units in computationally interesting ways, like the waterfall assembly algorithm of Section 4.3.3. In this section, a similar graph assembly task for C/E units is accomplished using the *turtle assembly* algorithm.

The more general and computationally powerful functionality desired from C/E units also adds complexities which do not exist when working with the structure and syntax of Boolean logic trees. As mentioned in the brief introduction above, there are several new challenges:

- the choice of a single, finitely-connected C/E net primitive, since a C/E net is constructed using two different types of nodes with potentially unlimited connectivity. Boolean logic has the advantage of a canonical structural form derived from direct electronic implementations.
- the assembly of arbitrary graph structures, requiring self-links and therefore distinguishable structural *identity*. NOR unit assembly was able to remain agnostic as to which unit instance was added to a growing tree, so long as the unit had the correct structure.
- a mechanism to control the output of a partially-completed C/E unit structure without running into the complexity bounds of predicting arbitrary C/E net output. In particular, the simple blinking properties of NOR unit structures described in Section 4.3.1 allows simple normalization of state across all NOR-units, but in general it is not possible to design a similar mechanism into C/E units without restricting the structure of intermediate assemblies.

Echoing the order of the previous chapter, the C/E net primitive designed for assembly is first introduced in Section 5.2. Using the localization construction, this C/E net primitive is shown to be sufficient for the construction of arbitrarily structured meta-C/E nets. Next, the CORAL unit which incorporates the C/E net primitive is described - the C/E unit (Section 5.3). Using the properties of the C/E unit, as well as additions useful for graph assembly, compact state manipulation, and structural verification (introduced in Section 5.4), the full turtle assembly algorithm can be described in Section 5.4.3. The remainder of the chapter is then dedicated to the complete C/E unit implementation (Section 5.5) and the example assembly of a computing unit.

5.2 C/E net primitives

The previous chapter describes an implementation of assembling units based on NOR operations. This is a useful and simple abstraction through which to understand and control the generation of meta-units (of interest in itself) but the NOR abstraction is placed on top of another - the C/E net controllers of the CORAL model. C/E nets were initially chosen as a useful language for assembly because of the distributed nature of Petri net systems, with built-in operations of synchronization and concurrency. These same properties remain useful when controlling large meta-structures extended across many units. Instead of regarding particular meta-structures as logical formulae, in many cases it would be nice to view these structures as a composition of C/E operations, organized like NOR meta-units into easily controllable chunks.

Of course, in the CORAL model this is already somewhat the case. Linked units with C/E controllers effectively become a larger C/E net, so that a large structure built from many identical units can be viewed whole as a single, combined C/E net with many isomorphic sections. Standard Petri net analysis techniques can be applied to the combined net, but it is not clear without further information what the various types of larger nets are capable of. Perhaps a particular unit type generates only uninteresting nets when linked to copies of itself, or only particular classes of nets. Ideally one would like to ignore unimportant operations of individual units and lift the description of the combined net to a higher level, so that what the large network is actually *doing* can be more easily understood.

The approach taken here is to design an individual unit which behaves like a small, generic part of a larger C/E net. Like NOR operations are used as primitives for building logical expressions, C/E primitive operations when linked to one another can build any C/E net. As a result, the behavior of assembling units emulating a C/E primitive may be described at a higher level than simply the combined C/E net formed by these units. This bears repeating, since the jumps in levels of description are hard to follow: the combination net of a structure formed by many atomic units emulating C/E net primitives, where the units themselves are implemented as atomic C/E nets, can also be described at a higher level as a *different* C/E meta-net. As nested NOR operations may be reformulated into equivalent logical operations (including meta-NOR units), C/E primitive operations in combination can be interpreted as equivalent C/E nets (including meta-primitives). Unlike logical operations, however, the class of C/E nets is Turing-complete when the size of the network is unbounded and zero-testing of places is possible (as it is with C/E capacity limits) (Peterson, 1981). In addition, the symmetry at multiple levels of description extends not only upward but also downward to the atomic unit itself, so that the atomic controller core no longer needs to be particularly atomic and might instead be a combination of primitives. This flexibility in levels of description is a novel property in a system which respects conservation of mass, and effectively divorces the computation from the scale at which it is created. Any behavior described as a C/E net can be built big, small, or in-between.

The above discussion assumes that a suitable C/E primitive exists which emulates the core functions of other C/E nets. Unlike the NOR operation for logical expressions, there is not a standard, composable Petri net primitive immediately available which requires only limited connections to other primitives. This perhaps reflects the general focus of Petri net literature, which seems to center on higher-level modeling applications and protocols. The idea of a concurrent

computing base (in the context of a basis for the laws of modern physics) was a focus of work by Petri himself (Zuse, 1969; Petri & Smith, 1987; Petri, 1996; Petri & Reisig, 2008; Petri, 2008), but, in his own words, was a "forgotten topic" with the rise of the ubiquitous Von Neumann architecture as a fundamentally synchronous basis for computers and the success of Petri nets as verification tools. A simple construct which allows the generation of computationally universal Quine transfer ($Q(a,b,c) = a\bar{b} \vee bc$, $a,b,c \in 0,1$) has been demonstrated by Petri (Figure B.1 in the appendices). The composed nets built from the construct are non-interactive without modifications, however, and the pairwise interactions of the CORAL model require extra transitions and ports for further assembly. Petri's construct and modifications for interactivity are explained further in Appendix B. A simpler primitive allowing more direct, interactive emulation of meta-level C/E nets is also possible to derive, however, by carefully examining the types of actions small portions of C/E nets carry out.

5.2.1 Designing a C/E primitive

As was defined earlier in Chapter 3, Petri and C/E nets are bipartite graphs of places and transitions where each place may be connected to any number of transitions and each transition may be connected to any number of places. In a C/E net, places may hold zero or one tokens which determine the transitions which are able to fire. The overall goal when designing a primitive for such a net is to be able to divide any net into small, identical pieces such that the overall operation of the network as measured from certain designated transitions is unchanged.

Since the net primitive will eventually be modeled as an atomic unit in the CORAL simulation, any part we choose must conform to basic CORAL constraints. The first constraint we discuss here is the bounded connectivity limit of CORAL units. The arbitrary connectivity of C/E nets is impossible to emulate using any type of CORAL atomic unit (and probably any realistic assembling unit), since the number of complementary ports must be finite. Choosing a primitive with any number of finite ports results in the unit being unable to directly emulate networks with higher connectivity. As a first step in creating any type of C/E primitive, upper limits must be set on the number of place-to-transition and transition-to-place edges. Only cyclic and chain networks are possible if this limit is set at 1 - a place may only connect to a single transition and a transition to a single place. A sensible, simple choice is a maximum of 2 incoming and outgoing edges per place or transition, which allows the partial emulation of more highly connected C/E net by converting more highly-connected places or transitions into tree structures of multiple branching places or transitions. There are some subtleties required when handling transitions with multiple incoming place edges, which will be described in more detail further in the section and chapter.

The second major constraint placed by the CORAL model on net primitives is the requirement that all interactions take place through shared transitions. Places are always internal to the units themselves. Assuming the limit on branching described above, which again requires a maximum of two incoming and two outgoing edges, the essential operations which must be supported by transition-bounded portions of a C/E net can be listed as follows:

- *produce*: producing a new token from one output transition
- *consume*: consuming a token from one input transition

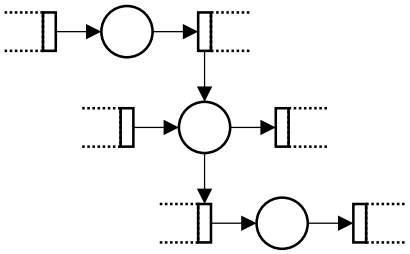


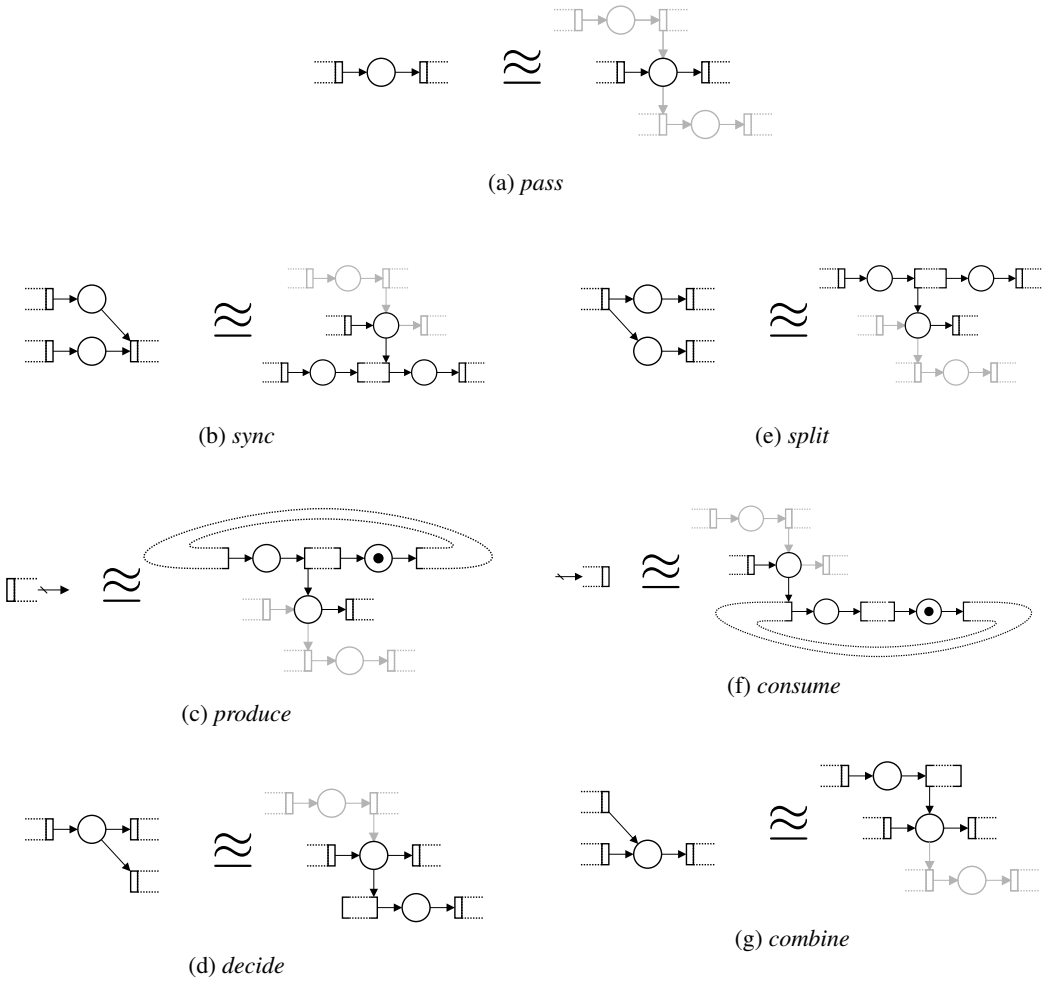
Figure 5.1: Small C/E net primitive. Compositions of this net with itself, by merging right transitions with those on the left in various combinations, can be made equivalent to any arbitrary C/E net. Dotted lines indicate these transition merge positions.

- *pass*: consuming a token from one input transition and then producing a token from one output transition
- *decide*: consuming a token from one input transition and then producing a token at one *or* another output transition
- *split*: consuming a token from one input transition and producing two tokens at two output transitions
- *combine*: consuming a token from one *or* another input transition and producing a token at a single output transition
- *sync*: consuming a token from two input transitions and only then producing a token at a single output transition

This list is also shown graphically in Figure 5.2. The seven operations above may seem arbitrary at first glance, but they can be regarded generally as permutations of the simple, feed-forward ways in which 0 to 2 transitions can be linked to 0 to 2 other transitions using 1 or 2 places. These choices will be further justified later with the introduction of the localization construction, which allows the emulation of other, more complex operations by combinations from this core group.

One trivial candidate for a C/E primitive is simply the raw combination of these basic C/E nets, though the CORAL unit implementation would be much more complex than necessary and require 16 complementary ports. Alternately, it is possible to collapse the full set of these operations into a few core types. For example, the *produce* and *consume* operations can be emulated by linking pass-through and splitting or combining operations, respectively, so they become unnecessary to include directly. Other operations can share input and output transitions, assuming non-connected input and output transitions not in use will not fire. By using this overlap, *decide* and *combine* can also be collapsed into linked *produce-sync* and *split-consume* operations. The final result is a much smaller 6-transition network composed of overlapping *sync*, *split*, and *pass* operations (Figure 5.1) which, when composed with itself appropriately, is able to emulate all other parts as shown in Figure 5.2.

One further operation is required for the *localization* of C/E nets, designated *pass&sync*. The addition is necessary to mimic highly in-connected transitions in C/E nets (representing a highly centralized action or operation) in a fully distributed way. The standard sync operator locks the places being synchronized such that if a second token never appears the first place marked will



(a) *pass*

(b) *sync*

(e) *split*

(c) *produce*

(f) *consume*

(d) *decide*

(g) *combine*

Figure 5.2: Essential operations supported by the C/E net primitive from Figure 5.1. Transitions linked by dotted lines indicate a merging of transitions between primitive units, not drawn as fully solid here to better distinguish each component. Each composition may depend on previous compositions defined in rows above. The looped transitions in (5.2c) and (5.2f) indicate self-connections, where tokens emerging from right output transitions of a C/E primitive are fed via another attached unit back to the left inputs. Grayed portions of the C/E primitive constructions are inactive and not connected to other net components. This is possible because CORAL port transitions are inactive when not connected.

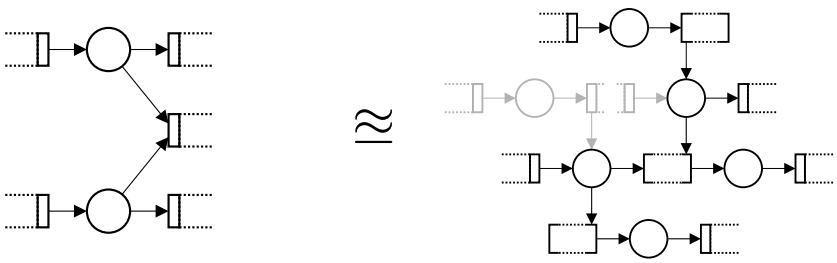


Figure 5.3: The final *pass&sync* operator, required for emulating transitions with many incoming place edges. Though the position of the top-center transitions may suggest otherwise, only three transitions are paired in the rightmost diagram (not the light gray transitions).

remain marked. With infinite C/E edge counts there is no problem, since a transition with many in-places can simply fire instantaneously and remove all tokens from all places simultaneously. The same behavior given only limited edges requires multiple levels of sync operations performed in a nested sequence to ensure that all required in-places have tokens. Much of the time the nested sync operations (each of which is only able to verify the presence of two tokens) will not complete because some tokens are missing from particular in-places. When this happens, the tokens must be passed *back* to the original in-places to allow other transitions to fire. As mentioned above, this cannot happen with the standard sync operator, which deadlocks in the single-token case. The *pass&sync* operator adds nondeterministic choice as an atomic action, and so allows a "guess" of whether all in-places have tokens to be wrong. Figure 5.3 illustrates the new operation, which is again simply a combination of the primitive shown in Figure 5.1.

It is important to note that the C/E net primitive used in these constructions has not been proven minimal. Given the pairwise constraints of the CORAL model, a lower topological bound of at least three linkable transitions are required for any primitive which builds complex graph structures. Two places are also required at minimum, since otherwise no transition, paired or otherwise, can link more than two places. This leads to combined C/E nets with no transition branching, a significantly limited class of C/E nets. So, if not minimal, the construction in Figure 5.1 is at least less than twice as large. A slightly simpler 4-transition version of the primitive is capable of emulating every operation above except *pass&sync*, and it is suspected that further simplification is not possible if this operation cannot be eliminated. A proven minimal unit would be interesting, however, and perhaps further refinements are possible in the future.

5.2.2 Localization

Given the C/E net primitive above, capable of emulating each feed-forward operation, the *localization* algorithm is now introduced as a way of partitioning arbitrary 1-safe Petri nets into linked copies of this primitive. As mentioned previously in Section 5.1, the basis of localization is conceptually similar to the NOR-based logical identities of Section 4.1 but uses C/E net primitives in place of a NOR operator. Every C/E net can be decomposed into linked 1-place networks, and groups of C/E primitives with identical behavior to these subnets can then be substituted.

The procedure intuitively works by slicing each shared transition into a copy for each place, creating for each place a separate, "island" network linked by *connections* to other islands. New islands are also created for sliced transitions with more than two edges in the previous network,

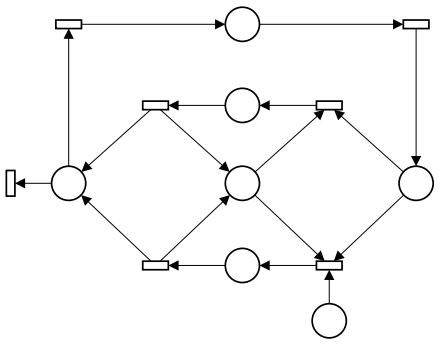
since only pairs of transitions can be assembled in the CORAL model. Finally, each island is normalized into a connected combination of islands containing primitive operations, which is always possible due to the simple single-place structure. These simplified islands may then be built using CORAL units emulating the C/E primitive.

Safe Petri nets, and *not* C/E nets, are the input to the localization process, which is an important distinction. C/E nets can be transformed into safe Petri nets (and trivially vice versa) via a simple construction described earlier in Chapter 3, so this does not limit the applicability of the process. Emulating networks with contact (blocking), like C/E nets, becomes challenging conceptually when places become linked distantly via intermediate primitives. Viewing the C/E net as a safe Petri net allows transitions to depend only on incoming markings, and the localized (and safe) result may then be interpreted again as a C/E net.

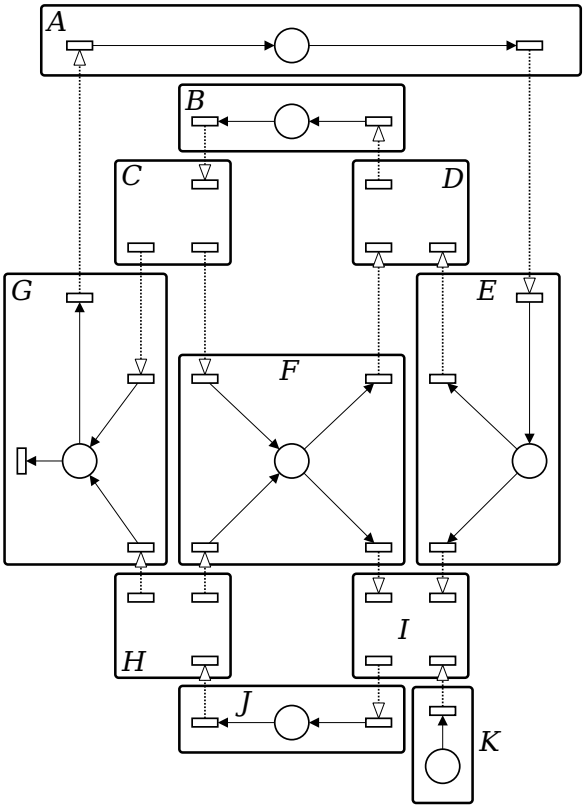
To begin, as mentioned above, a safe Petri net is divided into a set of *islands* derived from the places. Each island contains a single place, as well as a copy of all attached transitions, found by following incoming and outgoing edges to and from that place. This divided net is of very simple structure, though the number of incoming and outgoing transitions may be large. A new construction is defined, the *island graph*, which contains a node for each island and a set of *connections* between islands attached to incoming and outgoing transitions. If a transition is shared by two places (and therefore two islands) there is a connection in the island graph between these two transitions. Where a transition is shared by many places, a new island is created with incoming and outgoing transitions complementary to the copies of the transition at each island. These transitions are also connected in pairs. Figure 5.4 shows a demonstration of the island construction on a sample graph.

The island graph as described above essentially corresponds to units in the CORAL environment, with connections between incoming and outgoing transitions. Assuming the original net had no transitions or places with many edges, the initial island graph is already a target structure, buildable via C/E primitives, which emulates the original net. In general this will not be the case, and further processing is usually needed to simplify more complex islands into equivalent connected primitives. The limitation of pairwise assembly also requires addressing, since this disallows transitions contained in more than two islands from merging. As per the construction, these are initially represented as new, empty islands containing only transition copies.

Figures 5.5, 5.6, and 5.7 illustrate constructions for the normalisation of island graph nodes into combinations of primitive operations. The first step in breaking nonstandard islands into net primitives is to normalize the transition-only islands. If there is only a single incoming connection, representing a transition which branches outward to many places, the island can be replaced with many branching copies of the *split* operation (Figure 5.5). If there are many incoming connections, however, the situation is slightly more difficult. As in the outgoing case, the limited branching nature of the net primitive means that transitions linked to many incoming places must be represented by a tree of *sync* operations. But, unlike the outgoing case, it is unclear whether the highly in-linked transition emulated by the transition island should have fired until *all* of the sync operations complete. For example, imagine a net where 3 of 4 places are marked and a transition fires only when all 4 contain tokens. If this in-linked transition is simply replaced by a nested tree of sync operations comparing pairs of places and then comparing the results, the first sync level



(a)



(b)

Figure 5.4: The island graph construction, starting from a sample 1-safe Petri net (5.4a) and resulting in an island graph (5.4b). An island (solid black borders) is created for each place, labeled with a letter, containing a copy of neighbor transitions and edges. Islands are also created for transitions with more than two edges in (5.4a), which contain only copies of each island transition (i.e. C, D, H, and I). Island connections are created between transition copies, indicated by dotted white arrows, with direct links created for transitions with two edges and corresponding links to transition-only islands for transitions with more than two edges.

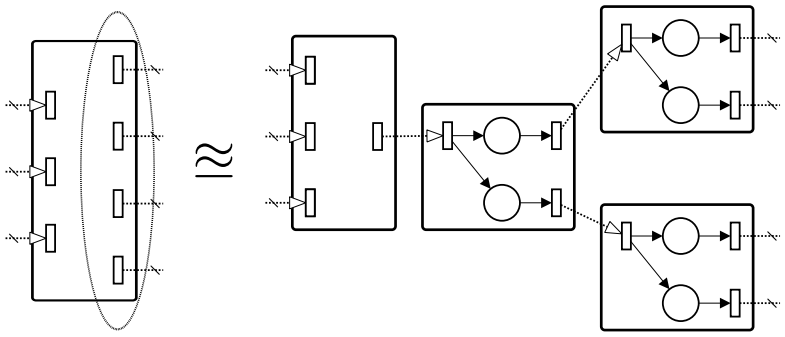


Figure 5.5: Transition islands with many outgoing transitions (circled with a thinly dashed line) may have those transitions replaced by nested *split* operations. *Pass* operations are also used to handle cases in which numbers of transitions are not powers of two. After the conversion, the new transition island has only a single output transition (though potentially many input transitions, handled by the construction of Figure (5.6)).

will remove the tokens from only the first pair of marked places and then remain waiting for the final token. This final token may never come, or may depend on other transitions with edges to the first pair of places which will now never fire. Essentially each sync operation is only a guess that the transition *might* fire, and these guesses must be allowed to be wrong. This is the reason for the *pass&sync* operation introduced above, which ensures that tokens removed for sync operations never remove tokens permanently unless all levels fire. A set of reverse transitions must also be added to incoming islands to support passing incorrectly *sync*ed tokens back to the original place. One reverse transition is placed in each place island with an incoming connection to the transition island (which changes the number of transitions in the place island but not the overall structure). The full construction is shown in Figure 5.6.

After these two steps, the island graph consists of normalized primitive operation islands, which have replaced the original transition-only islands, connected to the original place islands. Some of the place islands may also contain additional incoming transitions. The structure of every place island is, as before, a set of incoming transitions and a set of outgoing transitions all connected to a single central place. (Places with both an incoming and outgoing edge to a single transition are not considered here since they complicate the algorithm somewhat and can be emulated by additional places.) The next step is to find any open outgoing transitions not connected to any other islands (which only occurs in place islands) and connect them to *consume* operations. These open transitions exist if the original Petri net contained transitions without outgoing places. Port transitions are disabled by default in the CORAL model if a port is not connected, so connected consume islands must be created for these open transitions to fire. There are no open incoming transitions requiring the same treatment, since the original net must be safe and transitions without incoming places would produce an unlimited number of tokens.

The final step is to transform incoming transitions into trees of *combine* operations and outgoing transitions into trees of *decide* operations. Figure 5.7 illustrates this process, which is straightforward because the original Petri net is safe and can have no contact. (Contact was discussed in Section 3.2.) After this processing, the island graph contains islands representing primitive operations connected to place islands with only a single incoming and outgoing transition (which is

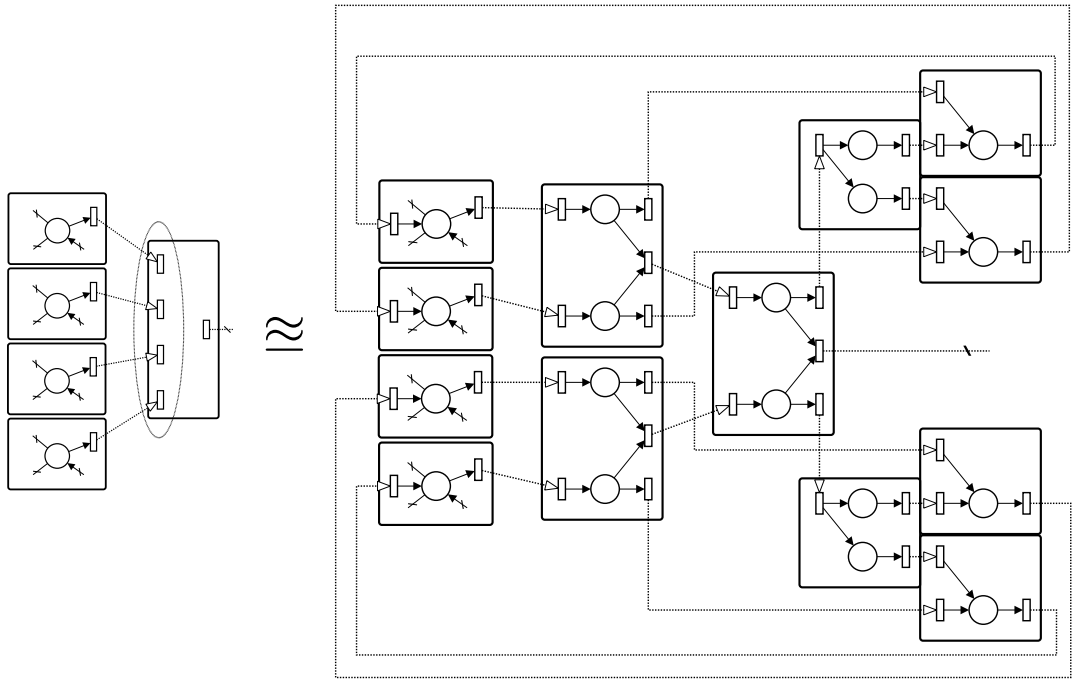


Figure 5.6: Transition islands with many incoming transitions (circled with a thinly dashed line) can be replaced by nested trees of *sync&pass* operations. As in Figure (5.5), *pass* operations are also used when numbers of transitions are not powers of two. When the central output transition of the *sync&pass* operations does not fire, indicating one or the other tokens may not be present, the tokens are returned via other primitive operations and an extra incoming transition to the original place islands. After this substitution there are no more transition islands, having all been normalized into connected islands of primitive operations.

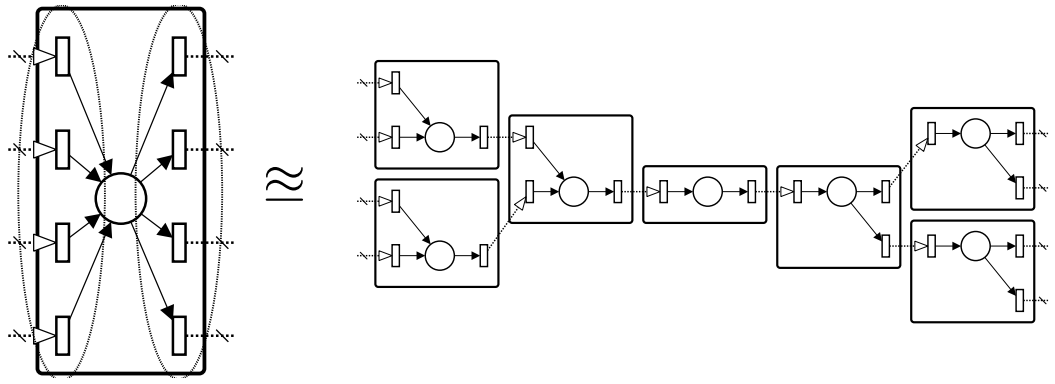


Figure 5.7: The multiple incoming and outgoing transitions of place islands (circled with a thinly dashed line) can be replaced by nested trees of *decide* and *combine* operations. The remaining root place island becomes a *pass* operation.

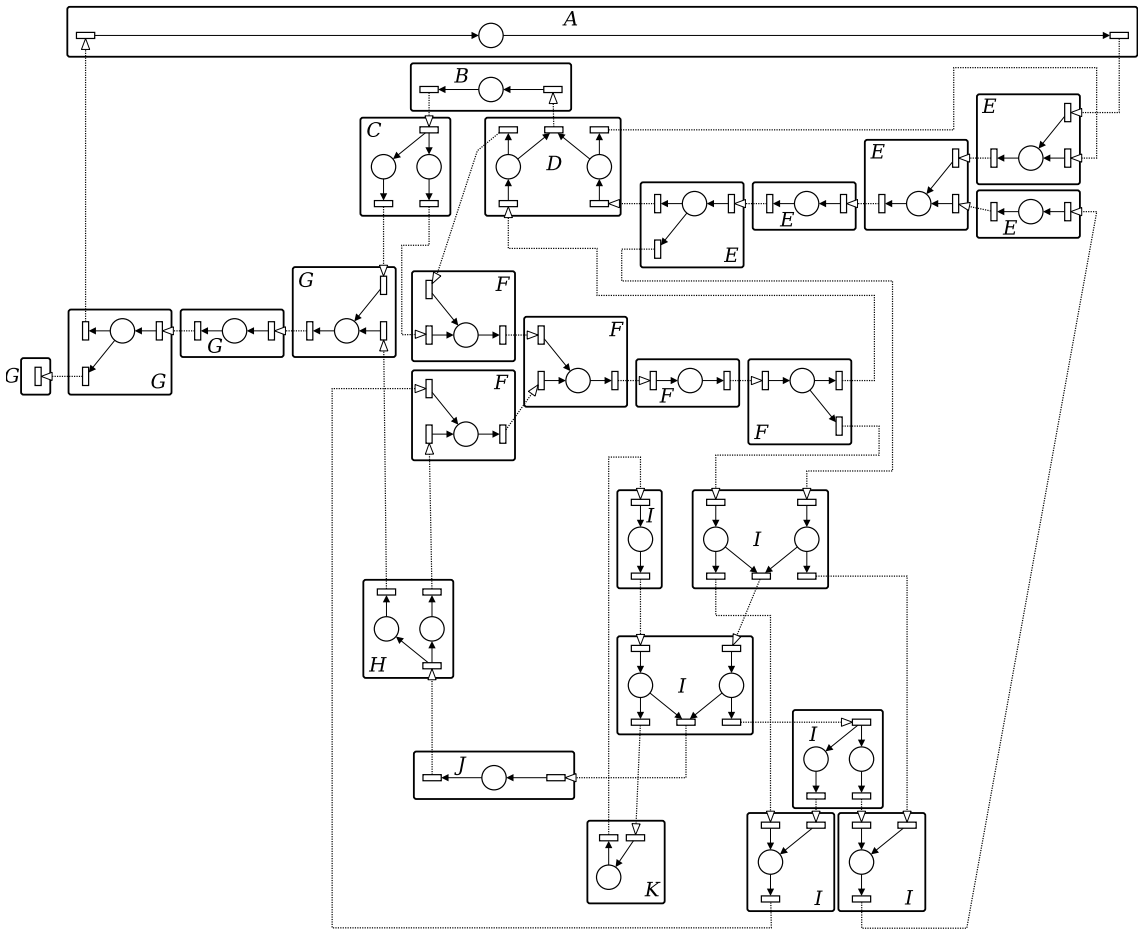


Figure 5.8: The final localized island graph, containing only islands of primitive operations. The labels on operations indicate the island of Figure 5.4b from which the operation islands were derived. This graph can then be used as a target graph for the C/E assemblers introduced in Section 5.3.

itself a *pass* operation). Slightly more efficient graph structures can be produced by allowing the original place islands one or two outgoing transitions (since these are also the primitive operations *decide* and *combine*) but doing so again complicates the algorithm description.

Figure 5.8 shows the final localized island graph derived from the example safe Petri net of Figure 5.4a. By combining transitions linked by connections, a localized Petri net is created. This net is safe if the original net is safe, since the primitive subgraphs representing the original transitions will only move tokens to other place islands if the original net could have fired the corresponding transitions. Therefore it can be implemented using CORAL units with capacity-limited C/E nets, which are defined identically to Petri nets when contact is not an issue. In addition, the behavior of the expanded net emulates that of the original net in that the incoming transition edges of the localized island structures fire only if the original net transition (from which they were constructed) might have fired. (Incoming *sync&pass* trees as shown in Figure 5.6 are a special case; the root of the tree construction fires as the original transition.) If the expanded net is attached at these transitions to some external device, and the central place of the root place-island *pass* operations set to the initial net marking, the potential orderings of transition firings

are identical to the original network. In other words, if the expanded net was actually built using CORAL units in a chemical bath, it could be "plugged" in at these transitions and act as the network it was derived from.

5.3 C/E assemblers

By designing an assembling unit which contains the C/E net primitive shown above in Figure 5.1 and also supports the graph construction mentioned in Section 5.4.2, arbitrary computing devices of any size and any function can, in theory, be constructed. C/E nets of $O(n^2)$ can simulate a linearly bounded automata of size n (Jones et al., 1977; Esparza, 1998), so if the potential assembly size is unbounded the automata simulated may approach the computational power of an unbounded Turing machine. In practice, of course, there are limits to the size of any assembled structure, though the CORAL model was designed explicitly to allow the control of assembly over orders of magnitude.

The main challenge remaining is to somehow link in a useful way the computation performed by partially completed structures with the new assembly that needs to occur, as was done with NOR units. One major difference between partially completed NOR structures and partially completed C/E nets is the previously-mentioned computational power of the units. Predicting the output of arbitrary C/E nets is PSPACE-complete (Esparza & Nielsen, 1994; Esparza, 1998), which contains and is widely believed a superset of the NP-complete complexity class. There are no known efficient algorithms to generally calculate how inputs will affect the full structure of a collection of C/E units. Like the partially-completed yet fully operational Death Star in Return of the Jedi, the incomplete C/E substructures are just as dangerous computationally as the target C/E net. A solution to this problem using a parallel control path is described in the next section.

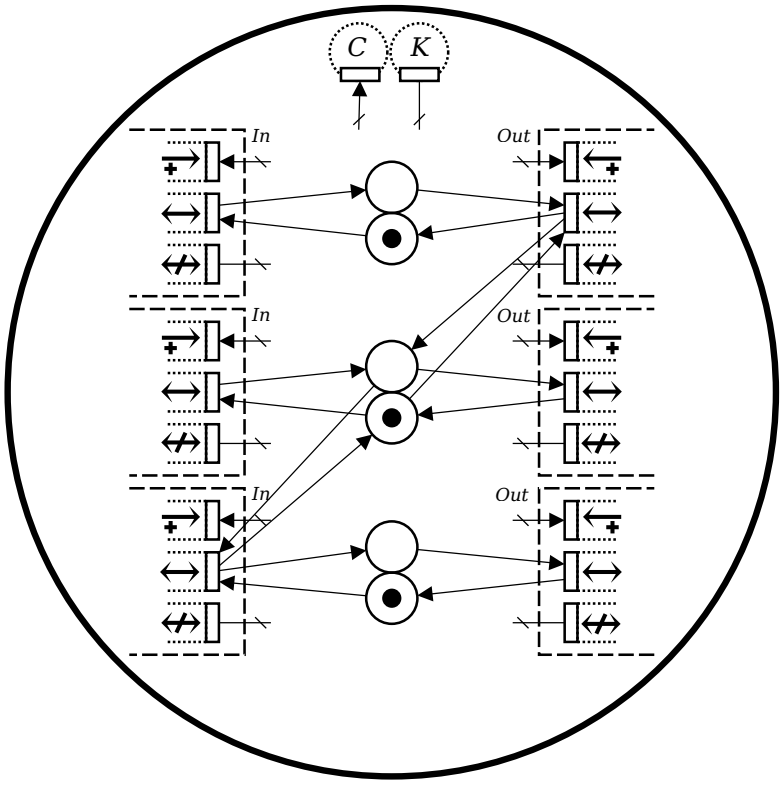


Figure 5.9: The core structure of a C/E unit. The C/E net primitive from Figure 5.1 is represented using pairs of places and synchronized transitions. Signals *C* and *K* allow one synchronized transition firing. Note that unlinking transitions are necessary in the C/E unit because of the graph assembly process.

Figure 5.9 shows an assembling unit containing the C/E net primitive, designated C/E units. Each of the three left and right linked transitions from the C/E net primitive have been designated as synchronized transitions assigned to an *In* or *Out*-port of the complementary *In* and *Out* types. The number of places has been doubled, creating complementary low, middle, and high places. When initialized with a token, these complementary places are constrained by the operation of the C/E net to always have a token when the regular low, middle, and high places do not, and vice versa. Transitions can only be enabled in C/E nets by places which contain a token, so the creation of place complements allows designating transitions that fire when the standard places are empty. This ability is useful later when controlling state changes and assembly operations. The NOR unit implementation shown as Figure 4.19 also used this doubling construction, since otherwise it would be impossible to enable particular transitions on a $false$ output. Extra linking and unlinking transitions assigned to each port are also needed for control of the assembly process.

There are two additional signal transitions in the figure which receive the signals *C* and *K*. Like the NOR units, C/E units may be clocked, and the *C* and *K* signals are used for the same purpose as the signals in the previous chapter. These transitions ensure that only a single C/E primitive operation (in the form of a synchronized transition) can occur after a *C* signal is sent, while a following *K* signal ends the potential for an operation and resets extra enabling places (not shown). Sending a repeated *CKCKCK*... background signal effectively removes this control,

allowing unrestricted execution. Since general C/E nets are not synchronized to a clock signal, this allows assembled meta-nets to execute in an unrestricted way while also allowing deterministic assembly. An additional signal could also be added which permanently (with a single signal) removes the clocking after assembly has occurred, though the simpler case is described here.

5.4 Assembly behaviors

C/E units do not have a single output value like NOR units, and instead store relevant internal state in the form of three sets of complementary places. As a convenient representation, each of these complementary pairs can be defined as set to 1 or 0 by a token at the higher or lower complementary place, respectively. These potential states, which can be represented as a three-bit string of high/mid/low (e.g. 010), are used to determine which, if any, of the link transitions will be opened. Like the NOR unit, a core feature of scalable assembly is that the logical state of the unit is linked directly to the assembly operation.

Each of the 3 *Out*-ports may be connected to any of the 3 input ports, making it necessary to assign 6 distinct token constellations to uniquely enable the link transitions of each *Out* and *In*-port. There are $2^3 = 8$ possible constellations of tokens representing the state of the C/E units, two of which are symmetric with respect to the ports (000 and 111). It is natural, therefore, to assign the three single-1 states (100, 010, 001) to open the corresponding *Out*-ports of the C/E unit, while the three double-1 states (011, 101, 110) open the *In*-ports of the C/E unit corresponding to the empty place. When assembly occurs, the *Out*-port link transition removes a token from the corresponding higher place while the *In*-port link transition moves the corresponding lower token to the higher place. As a result, the *Out*-port connected unit always has a state of 000 after assembly, while the *In*-port connected unit always has a state of 111. The C/E unit implementing this logic is shown as Figure 5.10.

Slightly more control is needed over assembly timing, otherwise structures form uncontrollably when internal computations required for graph assembly are taking place. This often leads to uninteresting, randomized behavior. Like the NOR unit priming signal, a special set of assembly clock signals *D* and *L* are required for units to begin assembly, after which the particular ports opened are determined by the C/E unit state. With the addition of the parallel *linear feedback shift register* (LFSR) control structure described below, these signals (and any additional others) could be removed and simulated via combinations of other LFSR signals, but are perhaps easier to describe and understand separately.

5.4.1 Linear feedback shift registers

To quickly recap, an assembling unit capable of building arbitrary C/E nets, the C/E unit, has been defined in the sections above. Onboard is a C/E net primitive, with additional logic that allows one to specify the opening of particular output ports or input ports in a way linked to particular unit/computational states. By manipulating these states appropriately and then allowing assembly to occur, it is possible to build structures from these units. Controlling the structures built still requires some mechanism of manipulating the states of the unit via background signals, which cannot be done directly through the computational inputs in C/E units because the devices built may be PSPACE-complete. A parallel state-shifting process is used instead, and to describe

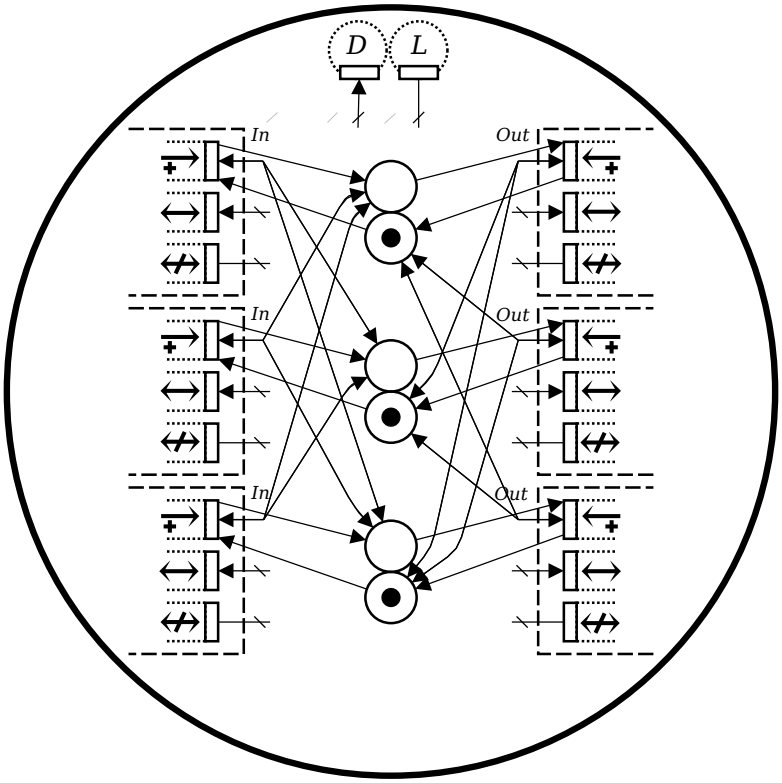


Figure 5.10: C/E unit with internal logic for enabling linking transitions. Each *Out* link transition is enabled by markings representing bit triples with a single 1 value in the corresponding place group, while each *In* link transition is enabled by a single 0 value in the corresponding place group. Multi-edges indicate both inward and outward edges between each source and target (places only link to transitions, and vice versa), and are used to group edges for clarity.

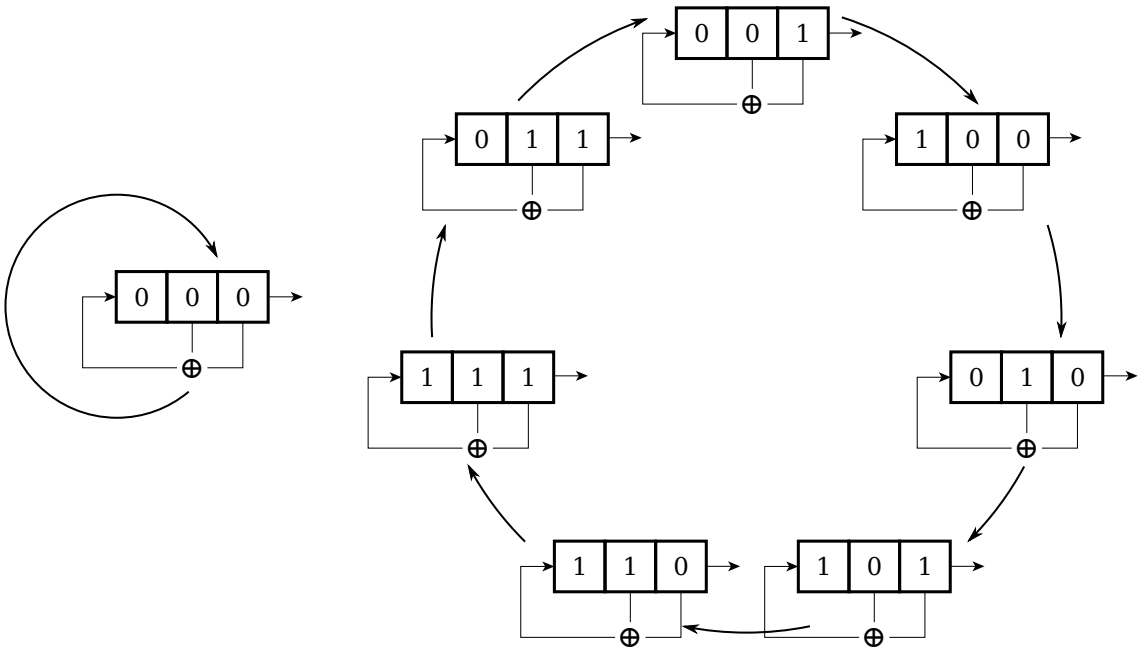


Figure 5.11: A 3-bit LFSR iterating through successive states. Arrows into and out of the shift registers indicate the direction of bit flow. The tap chosen here is maximal, meaning that the number of states iterated is $2^n - 1$ (where n is the length of the shift register) when the initial state is not all zeroes.

this process it is first necessary to introduce a device and algorithm commonly encountered in electronics and cryptography called a *linear feedback shift register* (LFSR). Essentially LFSRs are a very simple way to rotate through long sequences of states, and this ability is used to to create a very general distributed state shifting mechanism for C/E units. Much of the background information in the next section is taken from (Ronse, 1982), (Kennedy & Gentle, 1980), and (Goresky & Klapper, 2009).

An LFSR, as commonly described, is a clocked device with a linear array of n bits, known as a shift register. At each clock step, the value of bits is shifted one position (except for the last bit), such that a bit at position i is moved to a position $i+1$. The new, first bit in the array is generated by an XOR ($\oplus$) operation of the previous final bit in the array with bits from other positions, called *taps*. If the taps chosen correspond to a primitive polynomial of degree n which is a factor of the polynomial $x^T + 1$, the LFSR of length n will iterate through T states given an appropriate initial state (Goresky & Klapper, 2009). The primitive polynomial is of the form:

$$1 + \sum_{i=1}^{n-1} q_i x^i + x^n$$

All q_i values are zero, except those corresponding to the taps, which are equal to one. For the example above in Figure 5.11, the single tap is set at position 2 ($q_2 = 1$), and:

$$x^7 + 1 = (1+x)(1+x+x^3)(1+\underline{x^2}+x^3)$$

Therefore the period generated is of length 7 given a non-zero initial state. Such a period is *maximal* for an LFSR of length 3. The interest in LFSRs for many applications stems from the

simple implementation and the need for only very small numbers of taps for maximal periods of long registers. In addition, the state sequences generated have useful pseudorandom properties which distribute bit values near-uniformly, which is often practical in cryptology applications.

State swapping

For purposes here, however, LFSRs are used as the basis of a way to manipulate the state of arbitrary number of units in parallel, using a minimum of signals. Given any type of unit, a CORAL environment consists of many of these units with internal state in the form of token constellations. Units in the same state are deterministically indistinguishable because they must respond in the same way to all signals (though stochastic transitions can break this symmetry), and so units may be grouped into state sets containing all units with the same markings. Manipulating units for assembly using background signals is an operation which, in general, moves units in each state set to some other target state while keeping each state set distinct. LFSRs guarantee the ability to shift states in this way because each clock step advances every state to the next in the ring - there is no possibility that two LFSRs in different states will advance to the same state.

For arbitrary manipulation of state sets, often it will be necessary to not only shift but also *reorder* the sets. For example, assume C/E units in states s_1, s_2 and s_3 include an LFSR implementation with a period-7 state loop of:

$$s_1 \to s_2 \to s_3 \to s_4 \to s_5 \to s_6 \to s_7 \to s_1 \to s_2 \to \ldots$$

Permuting these states may be necessary for assembly so that $s_1 \mapsto s_2$, $s_2 \mapsto s_1$, and $s_3 \mapsto s_3$. Clearly, simply shifting states will never result in this mapping, since s_1 and s_2 always shift relative to one another. By adding an additional *swap* operation to an LFSR, however, which exchanges any two states, it is possible to generate any such permutation. All states may be advanced to any other, and so any pair of states can be chosen for the exchange. If we assume for the above example that we are able to swap s_1 and s_7, we can generate the target mapping using the operations illustrated in Table 5.1.

Operation	s_1	s_2	s_3	$s_1 = 001$	$s_2 = 100$	$s_3 = 010$
shift	s_2	s_3	s_4	100	010	101
shift	s_3	s_4	s_5	010	101	110
shift	s_4	s_5	s_6	101	110	111
shift	s_5	s_6	s_7	110	111	011
shift	s_6	s_7	s_1	111	011	001
shift	s_7	s_1	s_2	011	001	100
swap	s_1	s_7	s_2	001	011	100
shift	s_2	s_1	s_3	100	001	010

Table 5.1: Example generating a target state mapping of $s_1 \mapsto s_2$, $s_2 \mapsto s_1$, and $s_3 \mapsto s_3$ using shift and swap operations. The states are first *shifted* to place target state groups into swap states, then *swapping* reorders the states before a final shift into the mapping target states. Sample shift register bit states are manipulated alongside the state variables to illustrate how the shifting and swapping work when implemented.

Because only reversing two states is required, the swap operation is simple to implement. Consecutive shift and swap signals applied to C/E units with LFSR and swap functionality can perform arbitrary permutations of the current states, since the two operations are sufficient for bubble-sorting. These state permutations are key to the assembly of C/E units and unit structures. In practice, three signals *S*, *W*, and *R* must be used to direct the shifting *(S)* and swapping (*W*) of C/E units because it is impossible in the CORAL model to distinguish deterministically between single and multiple signals (e.g. SW vs. SSW vs. SSSW), as was also seen with NOR units. *R* is used as a reset signal for both *S* and *W* and also for signals defined below. A universal reset signal could also be designed which would also replace the clock-reset *K*, but for clarity the signal sets remain separate in this thesis.

State merging and recovery

A final operation, state *merging*, is also required so that C/E units in different state sets can be merged to a single set. Merging is necessary when one wants to execute a non-invertable state mapping, with many states mapped to the same target state. Swapping and shifting state values cannot generate these types of mappings, since by design the swap and shift operations never result in two different states assuming identical states afterward.

To merge states, some merging state is transformed into a target state without the target state being changed. One good choice of target state, in general, is the LFSR all-zero state. After being transformed, the all-zero state will not be affected by subsequent shifting or swapping, and so multiple merges can occur consecutively and intertwined with the other operations. In an opposite *recover* operation, the all-zero state is transformed back into the merging state chosen earlier. By shifting and merging particular states down into the all-zero state, then recovering the all-zero state back into the merge state, C/E units which are members of different state sets can be manipulated

to have the same state. An example state mapping $s_1 \mapsto s_2$, $s_2 \mapsto s_2$, and $s_3 \mapsto s_3$ which requires merging s_1 and s_2 while preserving s_3 is shown below in Table 5.2, where s_1 is the merge state and s_z is the all-zero state. As in the swap example above in Table 5.1, a period-7 LFSR is assumed.

Operation	s_1	s_2	s_3	$s_1 = 001$	$s_2 = 100$	$s_3 = 010$
merge	s_z	s_2	s_3	000	100	010
shift	s_z	s_3	s_4	000	010	101
shift	s_z	s_4	s_5	000	101	110
shift	s_z	s_5	s_6	000	110	111
shift	s_z	s_6	s_7	000	111	011
shift	s_z	s_7	s_1	000	011	001
shift	s_z	s_1	s_2	000	001	100
recover	s_1	s_1	s_2	001	001	100
shift	s_2	s_2	s_3	100	100	010

Table 5.2: Example generating a target state mapping of $s_1 \mapsto s_1$, $s_2 \mapsto s_1$, and $s_3 \mapsto s_3$ using *shift*, *merge*, and *recover* operations. As in Table 5.1, sample shift register bit states are manipulated alongside state variables to illustrate the operations.

It is possible that only a subset of permutations and non-invertible mappings are sufficient for the particular assembly operations needed to build C/E structures. While potentially non-minimal, the simplicity of the above approach allows more direct planning about units in particular states transforming to others. Any deterministic approach must require at least two operations to permute state sets because of the cyclic or chain topology of a singly-connected state diagram. Additional operations to merge and recover state sets from the all-zero state are required for the operation of C/E units, because the all-zero state corresponds to the all-low state required in C/E unit computation. The merge and recover operations can also be combined with swap, so that a swap first moves a state into the all-zero position and a recover moves it out. Figure 5.12 shows FSM representations using both the original and merged operations for a three-bit LFSR.

As a result, the shifting *(S)*, swapping *(W)*, and recover *(E)* mechanisms of state manipulation require very low numbers of signals to implement (including *R*) for the amount of flexibility they allow when manipulating the state of units. Using these signals, the states of any number of units can be manipulated in arbitrary ways. This flexibility is especially useful when filtering incorrectly linked units by propagating signals through a partially-built structure. Figure 5.13 illustrates simple LFSR logic added to the C/E unit, though in the full implementation, described in the next section, additional controllable state is needed.

Arbitrary distributed state manipulation

To summarize, orthogonal to the logic implementing a C/E net primitive in the C/E units is the LFSR logic described above. The LFSR is used to change the values of the pairs of places used

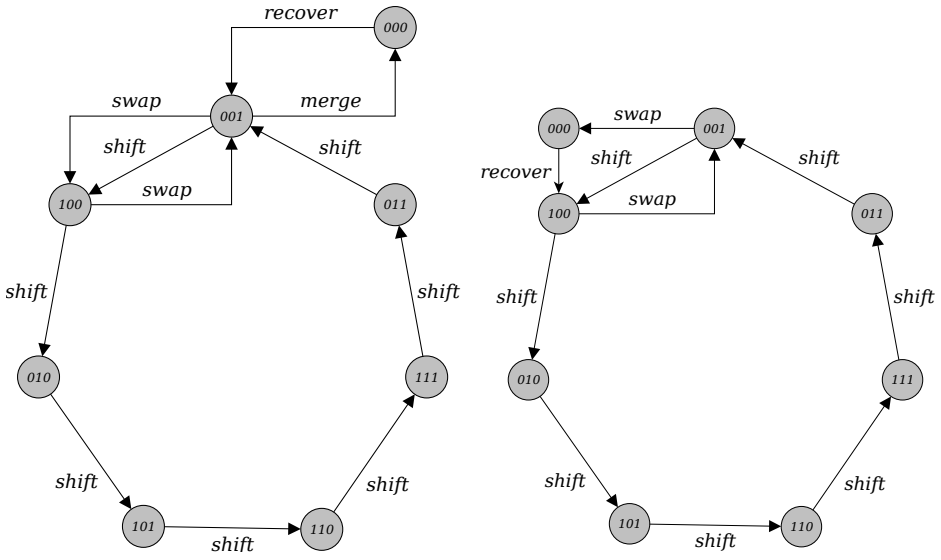


Figure 5.12: FSMs modeling the operations of the C/E unit LFSR state shifting. The FSM of (5.12a) requires entirely separate signals for merge and recover, while the FSM in (5.12b) has combined the operations so only a single additional signal is needed. Standard swapping is still possible, but a second *recover* signal must be sent after *swap* to reverse the states. Likewise, *merging* is just a single *swap* signal with the side effect of moving another state.

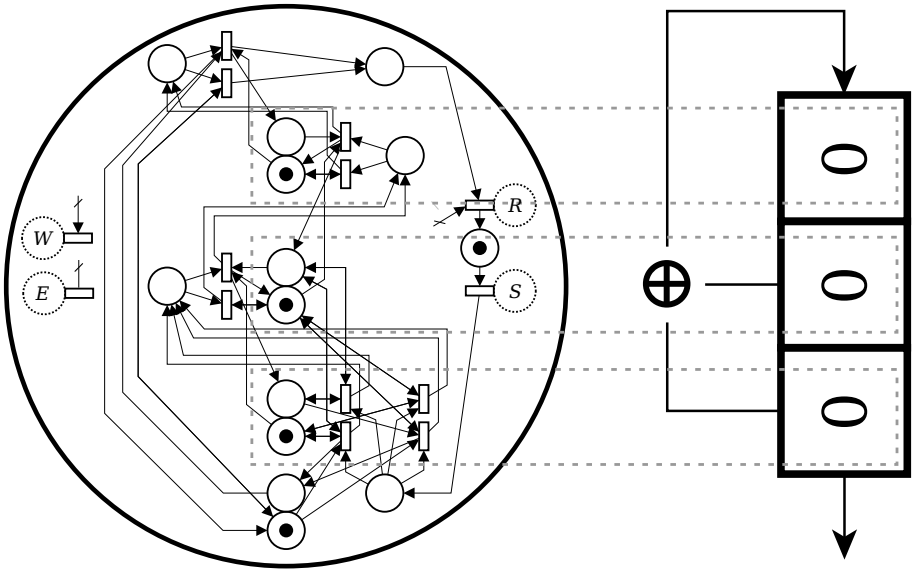


Figure 5.13: C/E unit with LFSR logic. The three complementary place pairs used by the C/E primitive (enclosed in the gray dotted boxes) can now be modified via the *shift* signal *(S)*. Full C/E logic for *swap* (*E*) and *recover* (*E*) not pictured. The reset signal *R* also resets places needed for these operations.

by the net primitive. These three place pairs represent the three bit shift register in the standard LFSR model, and the markings of the complementary pairs can be changed in arbitrary ways by sending signals for the four (or three, when combined) LFSR operations described above. Given the two components of the LFSR and C/E net primitive, it is possible to assemble structures by manipulating the port enablement via the LFSR components, and these structures can then pass tokens via the C/E net primitive to other parts of the structure. However, more is needed to create structures representing more complex C/E networks: filtering for graph structures containing cycles, and new markings for control over unlink transitions. These additions rely as much as possible on the existing assembly capabilities and C/E primitive logic, but fundamentally require extra, controllable states if deterministic assembly is the goal.

5.4.2 Graph assembly

Though seemingly a simple extension to tree assembly, the assembly of general graph structures using the CORAL model, or any well-mixed stochastically assembling system, is challenging. Tree structures, which by definition contain no cyclic edges, may be assembled deterministically from a single seed unit by repeatedly adding single units. Assuming an environment contains only incomplete tree structures and single units, opening a single port on a single unit in the tree structures and a single port on the individual units results in randomized pairing in which the particular unit that attaches to a structure does not matter. However, if the target structure contains a cycle, at some point in the assembly process a structure will need to open two complementary ports of two component units and somehow ensure that these units only attach to each other. The *identity* of which parts connect is important.

The CORAL model does not allow units to influence in any way the particular choice of port pairing, since units are assumed mixed beyond their control. As in an organic chemical reaction, where multiple products are unavoidable results in the formation of large chemical structures, the result of opening two complementary ports on a single type of structure will be chains and loops of this structure in various lengths. This situation is illustrated in Figure 5.14. Ideally only the target self-connections would form, but this result is highly improbable and becomes much more so as the number of structures becomes large.

One solution used in chemical engineering is to successively filter the reaction "products" until only a single type of structure remains. The only input mechanism in a CORAL environment are the background signals, so some set of background signals must be sent which tend to unlink the ports of chain and loop structures. After several assembly and filtering inputs, structures will be self-linked with higher and higher probability. Because identical structures are distinguishable only through symmetry breaking and chain and loop structures consist of identical copies of a sub-structure, all filtering methods must necessarily be stochastic.

The filtering algorithm used here, for simplicity, is tightly linked to the operation of the assembling unit itself, and so is described in more detail later in the section. As a brief summary, however, the signals sent by the algorithm can be thought of as a computational analogue to "shaking" the generated assemblies vigorously. Self-linked structures will always have a single part that shakes in the same direction, but the multiple sub-structures composing the chains and loops of multi-linked units may shake in different directions, causing the structure to break at these points.

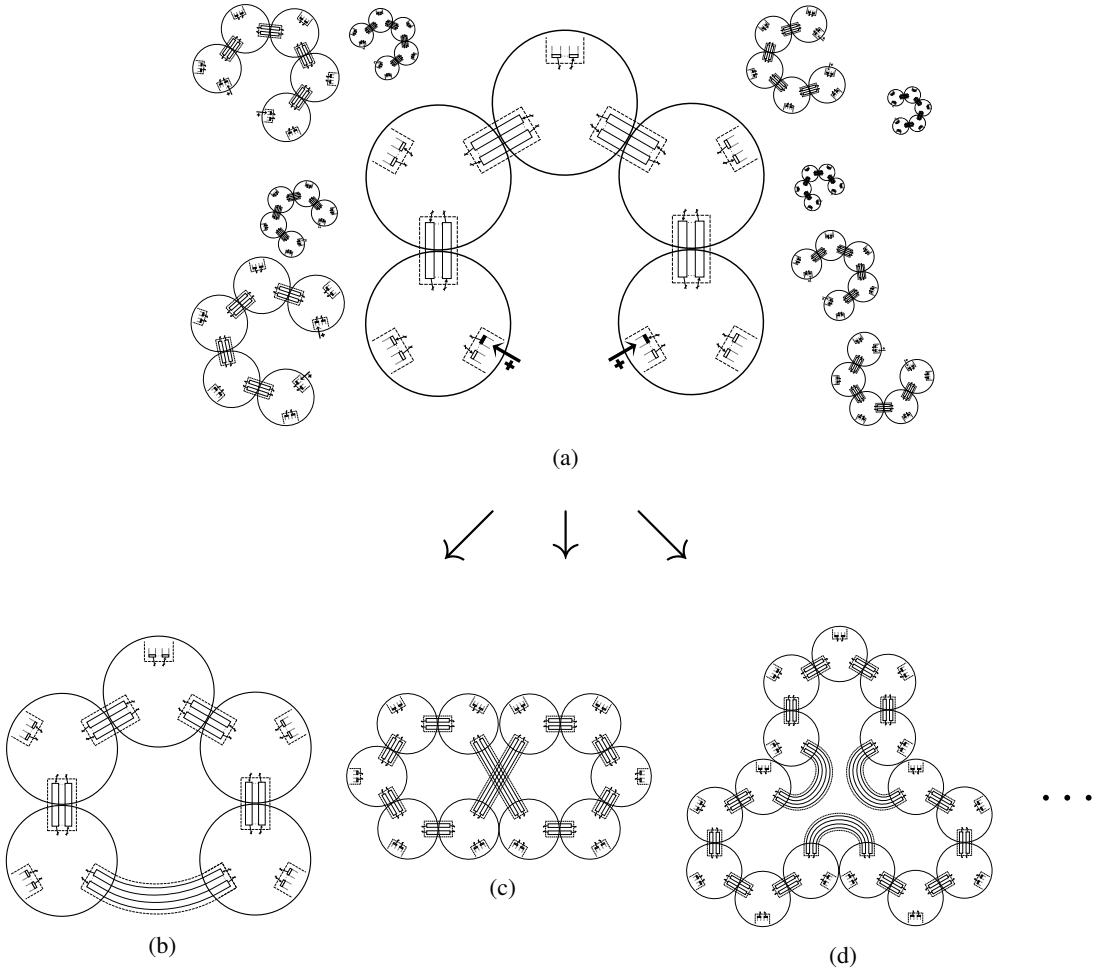


Figure 5.14: Two complementary open ports on many copies of an initial assembled structure (5.14a) lead to many different results. Not only self-connections occur as in (5.14b), but also many different sizes of looped structures appear (5.14c and 5.14d). Since there is no unit interaction possible before assembly in the CORAL model, no precautions are possible which prevent these extra "reaction products" from forming.

The next section introduces the primitive computational unit which is capable of these graph assembly computations and becomes Turing-complete when combined into graphs.

5.4.3 Turtle assembly

It is now possible to define the primary assembly mechanism for C/E units - "turtle" assembly - given the logical components introduced above:

- Petri net localization into C/E primitives,
- LFSR state shifting,
- and graph assembly and verification.

The name of the assembly algorithm is inspired by turtle graphics, where a single point or "turtle" forms an image from the trail of its motion (similar to the children's Etch A Sketch toy). At a high level, the core of the algorithm proceeds by progressively moving a unique turtle state through the C/E units of an incompletely assembled structure, using the operations of the C/E net primitive in combination with LFSR state shifting to select particular paths. The same turtle is also used to verify self-connection in graph structures by a symmetry-breaking decision to traverse or not traverse the newly formed cycle.

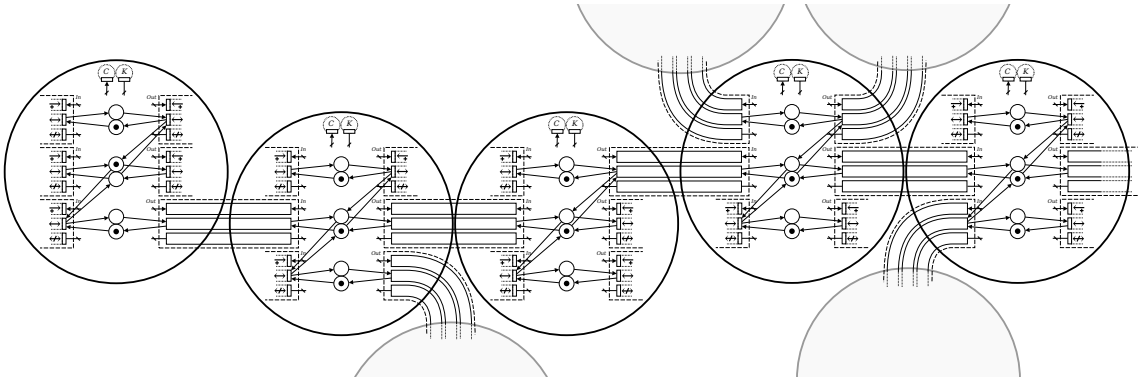
Starting from a large, partially-built structure of C/E units, the core of the mechanism works as follows. In a large structure, many units in the structure must necessarily be in the same states. In fact, most must be, since the structure size can ideally grow without bound while the number of states is limited. To be precise, we can say that a large structure consists of n units, each of which is in a single state $s_i \in S$. Since the number of units is much larger than the number of potential states ($n \gg |S|$), at least one state must be shared by multiple units.

We also assume the existence of a unique turtle state $t \in S$ at one of the units, which we would like to "advance" along some connected path of units through the structure (starting from the current unit). The exact value of this state is unimportant, but in order to allow the manipulation of individual units using this turtle it must always remain unique. To advance the turtle state to the next unit in the path, it suffices to put the next unit into some other unique state (since LFSR state shifting can always permute states if necessary). Assuming the path is long, and there are only finite states, the unique previous turtle states must also be merged into states shared by other units.

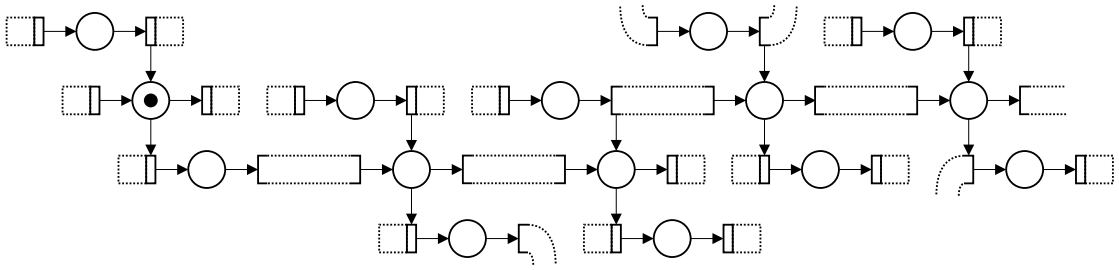
Figure 5.15 shows various representations of a path through a C/E structure. The compact labeled representation, Figure 5.15c, will be primarily used in the following discussion, but it is only a shorthand for the full C/E unit interactions. We assume a path of connected units (assuming, for now, only forward *Out*-port to *In*-port connections), and we represent such a path as a list of labeled states:

$$
\begin{array}{ccccccccc}
t & & s & & s & & s & & s \\
0 & & 0 & & 0 & \searrow & 0 & & 0 \\
1 & & 0 & \rightarrow & 0 & & 0 & \rightarrow & 0 \quad \cdots \\
0 & \nearrow & 0 & & 0 & & 0 & & 0
\end{array}
$$

The first unit in the path has the unique turtle state t, other units are in the shared structural state s while connections between units are represented by arrows. Since all *Out*-ports are complementary to all *In*-ports, arrows may connect different port positions (such as top to middle, middle



(a)



(b)

t	s		s		s		s	
0	0		0	↘	0		0	
1	0	→	0		0	→	0	⋯
0	↗ 0		0		0		0	

(c)

Figure 5.15: An example path through a C/E unit structure, shown equivalently as connected C/E units (5.15a), representative C/E primitives (5.15b), and a linear path of labeled states (5.15c). Empty grey units in (5.15a) and partially-connected primitive transitions in (5.15b) indicate other portions of the structure ignored for this particular path. The goal of turtle movement along this path is to place the end unit into a unique turtle state given a unique start unit turtle state.

to bottom, etc.). Multiple types of shared structural states are also possible, but not pictured. Units *not* on the path are assumed to be in structural state s as well, and do not (yet) affect path traversal so the connections to these units are not drawn. The goal is to transfer the unique turtle state to the end of the path (here dropping the particular state values because they are unimportant so long as unique per-label):

$$s \nearrow s \rightarrow s \searrow s \rightarrow t \cdots$$

The two tools available to perform this transfer are LFSR state shifting and individual C/E net operations. Permuting the original state path using LFSR operations cannot map a *particular* unit in s state to t, since each of the original s states are identical and therefore deterministically indistinguishable. Likewise, simply allowing C/E computation by sending the clock signals *CK* will do nothing, since the starting unit has edges only to disconnected transitions from the central place(s), as can be seen in Figure 5.15. By combining the two methods together, however, arbitrary transfer of turtle state is possible.

An example using the path from the Figure 5.15 the helps to clarify how the process works. First, the turtle and structure states are permuted such that the turtle state may *send* a token and the adjacent structural state may *receive* a token over a shared synchronized transition. For the first step, this mapping is $010 \mapsto 001 = t$, $000 \mapsto 000 = s$, where the labels s and t are reassigned to the mapping result. After state shifting is complete, t is still unique and s is identical:

$$\begin{array}{ccccccccc}
t & & s & & s & & s & & s \\
0 & & 0 & & 0 & \searrow & 0 & & 0 \\
0 & & 0 & \rightarrow & 0 & & 0 & \rightarrow & 0 & \cdots \\
1 & \nearrow & 0 & & 0 & & 0 & & 0
\end{array}$$

Next, a *CK* signal is placed in the environment and received by every unit, which allows a synchronized transition between the first and second unit in the path to fire, resulting in:

$$\begin{array}{ccccccccc}
t' & & s' & & s & & s & & s \\
0 & & 0 & & 0 & \searrow & 0 & & 0 \\
0 & & 1 & \rightarrow & 0 & & 0 & \rightarrow & 0 & \cdots \\
0 & \nearrow & 0 & & 0 & & 0 & & 0
\end{array}$$

After this single firing (or calculation step), the first and second units' states change to new values labeled s' and t'. In the current case s' is unique and $t' = s$, but in general this will not always be true. A trivial state permutation and relabeling is needed ($000 \mapsto 000 = t' = s$, $010 \mapsto 010 = s' = t$) to fully move the unique turtle state to the second unit:

$$s \nearrow t \rightarrow s \searrow s \rightarrow s - \cdots$$

A CK signal can again be sent, resulting in:

$$\begin{array}{ccccccccc}
s & & t' & & s' & & s & & s \\
0 & & 0 & & 0 & \searrow & 0 & & 0 \\
0 & & 0 & \rightarrow & 1 & & 0 & \rightarrow & 0 & \cdots \\
0 & \nearrow & 0 & & 0 & & 0 & & 0
\end{array}$$

Again, a trivial permutation changes s' and t' to t and s, respectively:

$$s \nearrow s \rightarrow t \searrow s \rightarrow s - \cdots$$

With alternating calculation and permutation steps, it is possible in this fashion to move the unique turtle state along forward-linked paths. First, state groups are permuted such that a single calculation in the correct direction can progress, and secondly the synchronized transition(s) are allowed to fire, resulting in new states for the turtle and adjacent structural unit. These new states are often, but *not* always, unique, however. For example, take the next step in half-traversed path above. Permuting the states appropriately ($010 \mapsto 100 = t$, $000 \mapsto 000 = s$), we have:

$$\begin{array}{ccccccccc}
s & & s & & t & & s & & s \\
0 & & 0 & & 1 & \searrow & 0 & & 0 \\
0 & & 0 & \rightarrow & 0 & & 0 & \rightarrow & 0 \quad \cdots \\
0 & \nearrow & 0 & & 0 & & 0 & & 0
\end{array}$$

After sending a *CK* signal, the output becomes:

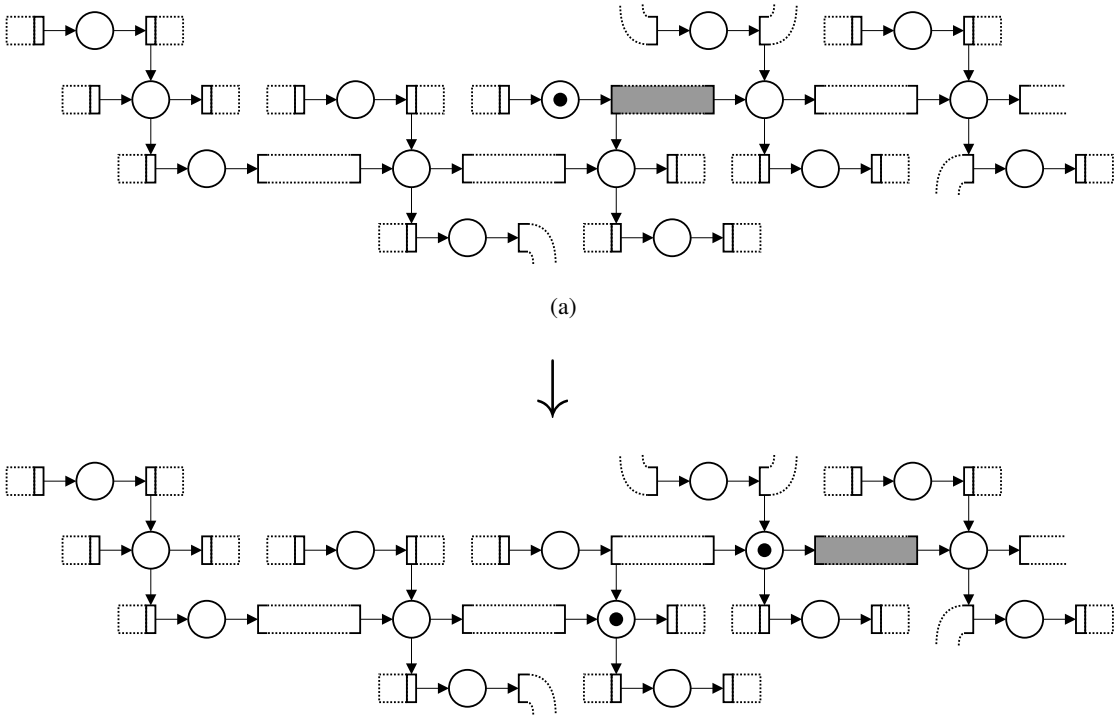
$$\begin{array}{ccccccccc}
s & & s & & t' & & s' & & s \\
0 & & 0 & & 0 & \searrow & 0 & & 0 \\
0 & & 0 & \rightarrow & 1 & & 1 & \rightarrow & 0 \quad \cdots \\
0 & \nearrow & 0 & & 0 & & 0 & & 0
\end{array}$$

Simple unit limitations

Because the topmost synchronized transition of a C/E net primitive outputs *two* tokens, as shown in Figure 5.16, the state of the next unit in the path (s') and the previous turtle state unit (t') are identical and now indistinguishable. Further path traversal can occur, but a unique turtle state is no longer guaranteed or possible (depending on other portions of the structure). Making things more difficult, most structures do not even have feed-forward paths between all units. Traversal of *backwards* connections must also occur (*In*-port to *Out*-port), and this is particularly the case for cyclic-connection verification paths, to be discussed shortly. Intuitively, to traverse backwards connections one permutes the s state to 100, 010, or 001 and t state to 000. With these active s states, however, *many* structural units (including those *not* on the path but in the s state) will be able to fire synchronized transitions in response to a *CK* signal, and unique state for t' or s' is impossible to ensure in all cases. Additional safeguards are needed to allow deterministic turtle motion in these cases, and extra state is required to represent these safeguards. The extra state also provides a natural basis for controlling unlinking transitions between units, a second major component of turtle validation, and the control of this state is discussed below.

Extra graph assembly state

Summarizing the discussion above, the control of unique turtle states between units of a structure is complicated when the turtle motion must traverse backward across connections or across synchronized transitions which output different numbers of tokens than they receive as input. Both of these cases result in the same general problem: there is no way of guaranteeing the uniqueness of states s' and t' after *CK* signals allow calculation (synchronized transition firings).



(a)

(b)

Figure 5.16: Traversing the example path from Figure 5.15 with the current turtle state at the central unit. Because of the nature of the path connections and C/E primitive computation, firing the synchronized transition(s) results in the next unit in the path and the central unit having an identical marking (5.16b).

To solve this problem (and add additional state to control unlink transitions), four extra places, representing two extra bits, are added to the LFSR implementation on the C/E unit. The extra bits do not affect the C/E primitive operation, and represent only a small addition to the LFSR logic because the same second-to-last tap position is maximal for a 5-bit as well as 3-bit register (allowing the LFSR to cycle through all 31 non-empty 5-bit states). The bit representation introduced above for the state of C/E units is extended for these bits by appending them to the front of the bit string, e.g. 10-101, 11-000, and so on. As with the 3-bit states, these 5-bit states may be permuted into any other using the shift, swap, and recover LFSR operations with corresponding environmental signals.

The primary use of these high bits is to allow and disallow the firing of synchronized transitions in particular directions. A C/E unit in state 01-### may fire only *Out*-port transitions (# representing any bit), while a unit in 10-### may fire only *In*-port transitions. The 00-### and 11-### states are interpreted naturally as all-disabled and all-enabled. Standard C/E or assembly operations do not change the values of the high bits, so that if all units are in 11-### states they may calculate normally. However, LFSR operations *do* modify the values of these bits, which lets unit states be shifted into non-interacting or partially-interacting states.

The directionality and disablement allowed via high bits allows more control over connections traversed for turtle motion. The state of the unit on the *Out*-port side of the connection is permuted to 01-###, while the state the unit on the *In*-port side is shifted to 10-###. Units in states irrelevant to the computation but necessary to preserve can also be safely shifted to unique "frozen" or inactive 00-### states. Assuming the lower bits are also set correctly and either the *Out*-port or *In*-port state is unique, only a single calculation between units in these states may proceed. The resulting states s' and t' are also distinct, since s and t differ in high bits and these do not change during C/E operations.

Example. The failed central unit traversal presented above now becomes possible using high bits. Each unit in structural state is allowed only to receive tokens from *In*-port connections, while the turtle state may only send tokens from an *Out*-port. Resuming the example of Figure 5.16 above, with the new high bits:

$$
\begin{array}{cccccccccccccccccccc}
s & & s & & t & & s & & s & & & s & & s & & t' & & s' & & s \\
1 & & 1 & & 0 & & 1 & & 1 & & & 1 & & 1 & & 0 & & 1 & & 1 \\
0 & & 0 & & 1 & & 0 & & 0 & & & 0 & & 0 & & 1 & & 0 & & 0 \\
- & & - & & - & & - & & - & & \overset{CK}{\to} & - & & - & & - & & - & & - \\
0 & & 0 & & 1 & \searrow & 0 & & 0 & & & 0 & & 0 & & 0 & \searrow & 0 & & 0 \\
0 & & 0 & \to & 0 & & 0 & \to & 0 & \cdots & & 0 & & 0 & \to & 1 & & 1 & \to & 0 \; \cdots \\
0 & \nearrow & 0 & & 0 & & 0 & & 0 & & & 0 & \nearrow & 0 & & 0 & & 0 & & 0
\end{array}
$$

The state change can be exploited using the LFSR shifting map since the high bits make *t'* and *s'* unique:

$$\begin{aligned} 10-000 \mapsto 10-000 &= s \\ 01-010 \mapsto 10-000 &= t' = s \\ 10-010 \mapsto 01-010 &= s' = t \end{aligned}$$

Finally resulting in $s \nearrow s \to s \searrow t \to s - \cdots$

Example. Reverse traversal of connections is also possible. For example, given a bi-directional path through a structure (linked only at the middle *In*- and *Out*-ports for simplicity), the high bit restrictions only allow a single synchronized transition firing between the unit in state t and the adjacent structural unit. Assuming states have been permuted initially for the first traversal:

t		s		s		s				t'		s'		s		s	
1		0		0		0				1		0		0		0	
0		1		1		1				0		1		1		1	
–		–		–		–		$\overset{CK}{\to}$		–		–		–		–	
0		0		0		0				0		0		0		0	
0	$\leftarrow$	1	$\to$	1	$\leftarrow$	1	$\cdots$			1	$\leftarrow$	0	$\to$	1	$\leftarrow$	1	$\cdots$
0		0		0		0				0		0		0		0	

Permuting the resulting states after the calculation:

$$\begin{aligned} 01-010 \mapsto 10-000 &= s \\ 10-010 \mapsto 10-000 &= t' = s \\ 01-000 \mapsto 01-010 &= s' = t \end{aligned}$$

Results in:

s		t		s		s	
1		0		1		1	
0		1		0		0	
–		–		–		–	
0		0		0		0	
0	$\leftarrow$	1	$\to$	0	$\leftarrow$	0	$\cdots$
0		0		0		0	

Unlinking operations

Unique state for unlinking operations also becomes possible with these two extra bits/places. Similar to the *In*-port/*Out*-port calculation restrictions above, assembly states are divided into *linking* and *unlinking* states via the high bits. Units in 01-### may enable *Out*-port *link* transitions, while units in 10-### may enable *In*-port *unlink* transitions. *Out*-port unlink transitions and *In*-port link transitions are always enabled given appropriate low-bit state/marking, but are restrained not to

fire by their port transition complement. Since assembly and calculation steps respond to different clock signals (C, K vs. D, L), the high bit states used overlap without problems.

Unlink transitions are enabled using states similar to those of link transitions. As with link transitions, three unique states are needed to identify the particular *In*-port unlink transition which should be detached: 10-100, 10-010, and 10-001. These correspond to the high, middle, and low *In*-ports, and are distinct from the link transition states given the additional high bits. When unlinking occurs, the three lower bits of the newly-detached source and target units are placed into the states which would be required for linking again. This is done for simplicity, since these states are already unique and often multiple assembly operations must detach/reattach units multiple times for validation of a cyclic connection. Choosing states this way allows the detached units to reattach to other units without additional state permutation.

Turtle verification

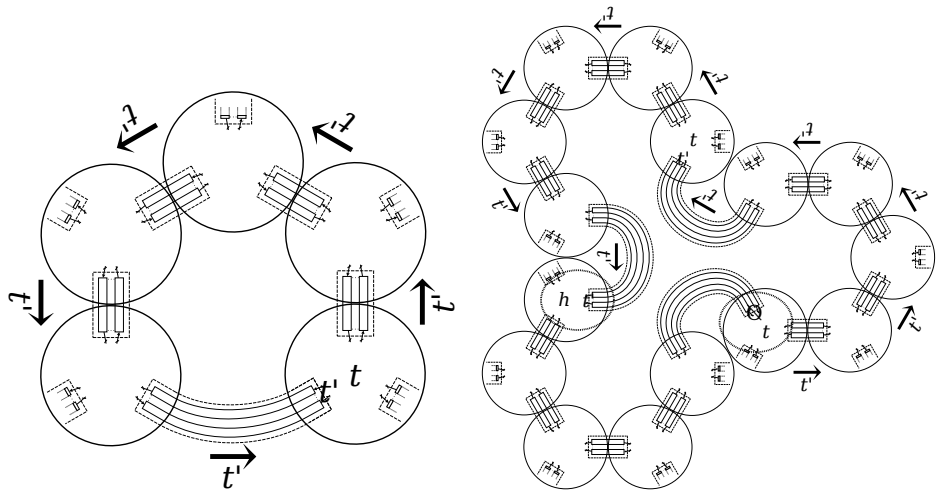
When a new connection is added to a partially-completed structure and creates a cyclic edge, the resulting structures must be verified for correctness. The basic procedure used, as was introduced in Section 5.4.2, is to ensure that the unit linked by the new connection is the same unit as the one found by traversing the rest of the cyclic connections in the opposite direction through the sub-structure. The idea is fairly simple but difficult to describe clearly in words, so Figure 5.17 provides a concrete example using the cyclic assembler from Section 5.4.2.

The turtle verification process begins when a cyclic connection is added to a growing structure. The target unit for this connection (*In*-port connected) performs a symmetry-breaking operation to place it in a holding state h or turtle state t. A path is then calculated through the *interior* of the sub-structure, starting and ending at the current unit position. If the structure is self-linked, this path will lead back to the same unit, but crucially the path leads to a *different* unit in a connected sub-structure if the structure is a chain or loop of multiple sub-structures. After traversing this path using the unique t' state (and shifting the h and original t states into frozen 00-### states), it is possible to compare the state sent around the sub-structure with the original state of a symmetry-breaking start unit. (The exact mechanism requires two lower-bit states which can receive the same token while maintaining individuality.) When t' is "received" and h was original start unit state, or nothing is received and t was the original start unit state, the states corresponding to these conditions are transformed into unlink states to remove the invalid connection between sub-structures. Link-enabling signals are again broadcast into the CORAL environment to reconnect newly-open ports, which may self-link or form chain and loop structures again.

Self-linked units can never have a mismatch in received turtle state, since the unit which "decides" to send the turtle message and later receives it (or does not) is the same. Therefore self-connections are stable under the turtle verification process. After several verification steps, chain and loop structures become less and less likely, and eventually are all destroyed.

Combined algorithm

With the LFSR state permutation, turtle motion, and turtle verification mechanisms described in detail, it is finally possible to sketch the full assembly algorithm for arbitrary C/E net structures: Algorithm 2. Overall it is similar in structure to the NOR waterfall assembly Algorithm 1, only replacing a tree traversal with a graph traversal and adding special handling for the creation of cyclic edges. In both, seed structures are created by a symmetry breaking operation of the original



(a) (b)

Figure 5.17: Two types of structures formed after a cyclic connection, one self-linked (5.17a) and the other connected as a loop of sub-structures (5.17b). Units on one side of the new connection "decide," via a symmetry-breaking operation into states h and t, whether or not to send a message (in the form of turtle state t') around the cyclic connection of the sub-structure. Using the turtle motion mechanism described in Section 5.4.3, the t' message state is advanced over the connections of each sub-structure simultaneously (possible because all substructures are identical). Eventually the message state reaches the original unit or the corresponding unit in a different substructure. Units which sent a message and did not receive one, or received one and did not send one (circled in a thinly dashed line) can then be manipulated via LFSR commands to unlink the connection. This mismatch never happens in self-connected units (5.17a), so they remain stable under verification.

homogenous CORAL environment of individual units, to which additional units are added one-at-a-time. Though the order of unit assembly is simply defined here, in practice it is useful to ensure that new nodes are attached in positions nearby the previous attachments and minimize turtle motion. In addition, a minimal spanning tree can be calculated so that the addition of cyclic edges and verification happens only at the end of the assembly process. The full C/E unit diagram data and turtle assembly source code which generates commands to assemble various structures is available online at:

- https://coralassembly.wordpress.com/

Again like the waterfall assembly algorithm, the pseudocode presented in Algorithm 2 encapsulates state permutation, turtle movement, and cyclic structure verification, each described above, as the functions `perm_states`, `move_turtle`, and `filter_structs`, respectively. All functions implicitly save the commands they generate to the `output_commands` variable. The `break_symmetry` command is not described elsewhere, but uses stochastic timing of transitions to change some units in a particular state to a new state. The core features of turtle assembly are outlined in `turtle_assembly` - a function which takes a multigraph structure with port information and implicitly outputs the commands necessary to build it using C/E units. *Localization* and the island graph construction (Section 5.2.2) are used to derive the multigraph structure and connected ports which emulate a desired safe Petri net.

The target multigraph structure with port information (`tar_struct`) is input to the function, and a `struct` multigraph is maintained as the current state of the growing structure. A mapping of labels to states is also maintained as the `states` dictionary, which is used and modified appropriately by core functions. All actual state information has been abstracted from the pseudocode, however, for simplicity and readability. The arbitrary labels given to the turtle state and frozen turtle state are `TURTLE_STATE` and `FROZEN_TURTLE_STATE`, the shared structural state is `STRUCT_STATE`, and a label for the individual stock parts is `PART_STATE`. These are implicitly permuted to appropriate values so that units will connect correct ports. Once connected, the number of filtering steps is specified by the constant `FILTER_STEPS`.

5.5 Implementation in 27 bits

The turtle assembly algorithm above is sufficient for the general assembly of graph structures, and through such assembly C/E unit structures can emulate arbitrary C/E nets. Figure 5.18 highlights the signals and main places of the final C/E unit, while Figure 5.19 shows the final C/E controller net including all signal and port transitions, consisting of 27 places and 49 transitions. Though this network looks quite complex embedded in a two-dimensional view it is effectively a superposition of all the smaller networks described above. A C/E unit of this design, when controlled by signals from the turtle assembly algorithms described above, is capable of forming structures which emulate any sort of C/E net, including its own controller. Notably this network requires fewer overall places and transitions than the NOR assembler, partially due to the efficient state-shifting LFSR algorithm.

Algorithm 2 Turtle assembly algorithm in Python pseudocode.

```
# Constants
FILTER_STEPS = ...
TURTLE_STATE, FROZEN_TURTLE_STATE = ...
STRUCT_STATE, PART_STATE = ...;

output_commands = [] # Stored commands

def turtle_assembly(tar_struct):


    struct = ... # Empty directed multigraph, current structure
    states = {TURTLE_STATE : ...} # All units initially in same state

    # Create our first partial structure by breaking symmetry
    break_symmetry(states[TURTLE_STATE], states)

    # Assemble C/E structure via graph traversal
    turtle = tar_struct.nodes[0]; struct.add_node(turtle)
    fringe = [(turtle, tar_struct.edges(turtle)[0])]
    while len(fringe) > 0:
        # Fringe is a list of node and next edge
        unit_a, edge = fringe.pop()
        unit_b = (edge[0] if edge[0] == unit else edge[1])
        cycle = (unit_b in struct)

        # Move turtle to new assembly point
        if not cycle: move_turtle(struct, [turtle, unit_b], states)
        if cycle:
           # If a cyclic conn, freeze turtle state at unit_a
           # and move to unit_b
           move_turtle(struct, [turtle, unit_a], states)
           move_turtle(struct, [unit_a, unit_b], states, freeze=True)
        turtle = unit_b

        # Permute turtle states and part states for assembly
        perm_states(states, ...); allow_assembly(states)

        # Filter new connection if necessary
        if cycle:
           for i in range(0, FILTER_STEPS):
              filter_structs(struct, edge, turtle, states)
              allow_assembly(states) # Reconnect unlinked ports
        else: struct.add_node(unit_b)
        struct.add_edge(edge)

        for new_edge in tar_struct.edges(unit_b):
           if not new_edge in struct.edges():
              fringe.append((unit_b, new_edge))

# Allows assembly, makes turtle state newly connected state
def allow_assembly(states): ...

# Core turtle assembly functions
def perm_states(states, mapping): ...
def move_turtle(struct, path, states, *props): ...
def filter_structs(struct, cyclic_edge, turtle, states): ...
def break_symmetry(state_to_split, states): ...
```

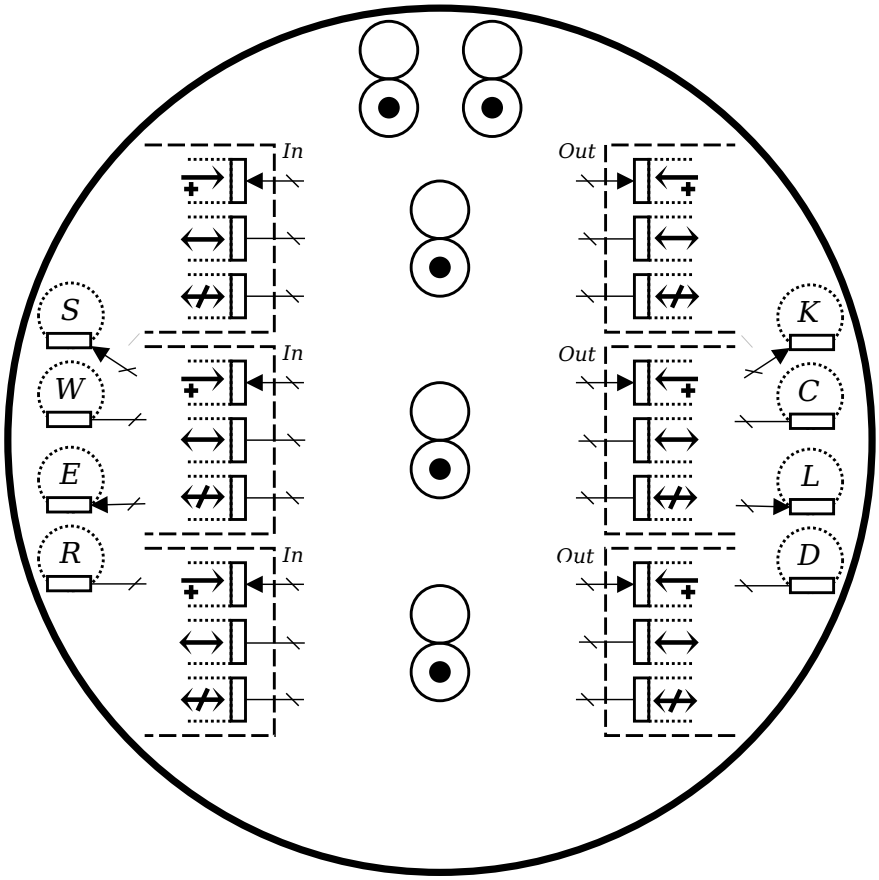


Figure 5.18: Diagram of the C/E unit with all port and signal transitions labeled. The three low pairs of C/E logic places and two high pairs of extra graph assembly places are also included. On top of this "skeleton," the C/E net primitive and LFSR logic are superimposed, resulting in the C/E unit implementation of Figure 5.19.

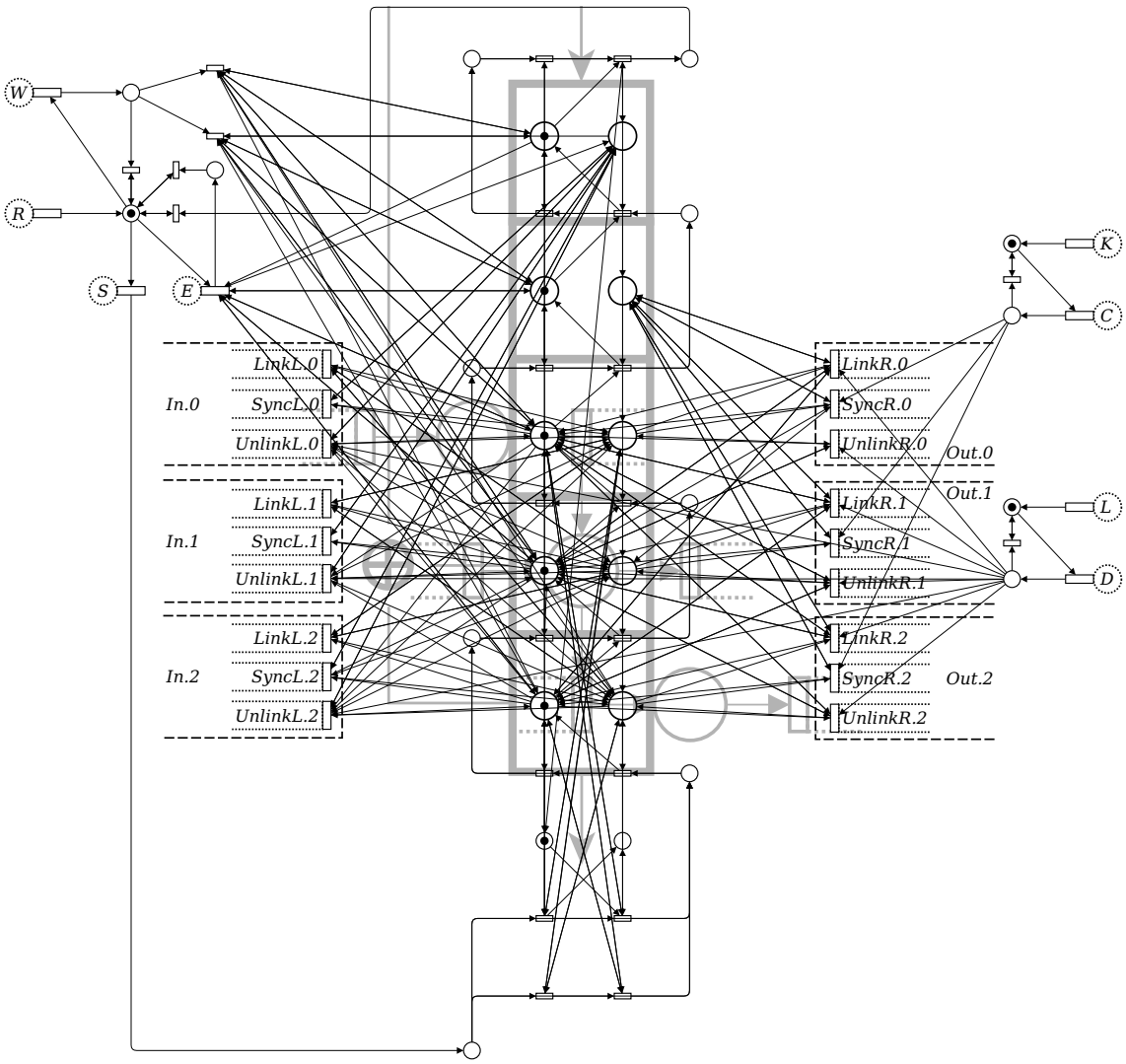
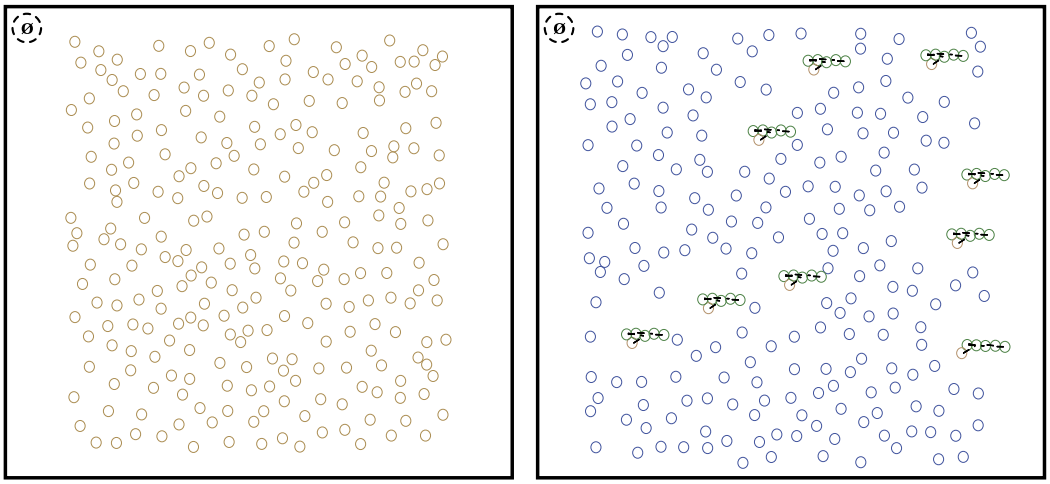


Figure 5.19: The full C/E unit implementation, with LFSR, assembly, and C/E net primitive logic superimposed. The actual graph in standard graph markup language (graphML) form is available with the source code, since viewing the full diagram this way necessarily obscures parts of the logic. Overall, the LFSR logic can be seen as the central zig-zag line with LFSR signals, while the assembly and net primitive logic share space in the many edges of the central area. Unlike the NOR unit assembler, no tracking of assembled edges need occur, which simplifies the port logic significantly.

5.5.1 Building a target network

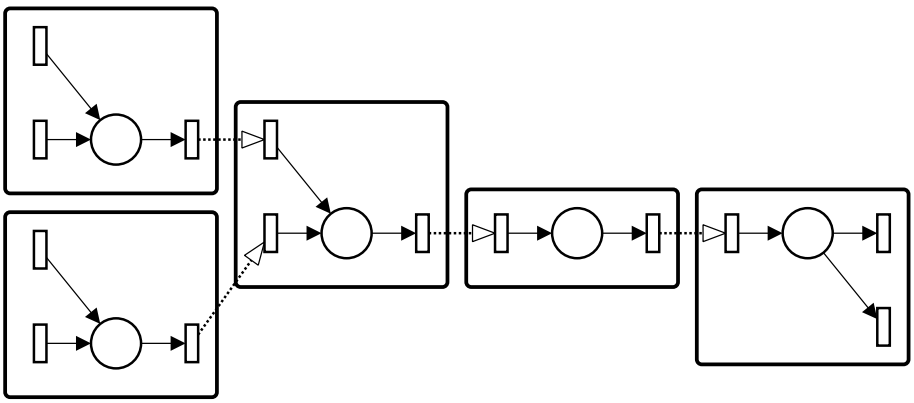
To demonstrate the assembly process, the implementation of Figure 5.19 is placed in a simulated CORAL environment and a target network is generated. As an illustrative example, the central *F* portion of the localized island graph from Figure 5.8 is built here. Assembling the full localized network is also possible, but the complex resulting graph is time-consuming to produce and difficult to view in 2D. Figure 5.20b shows the *F* islands transformed into islands containing only the C/E net primitive (using the identities from Figure 5.2), and this structure may then be built in simulation via turtle assembly. While the original place island contained no cyclic connections, the transformed result does (due to the *produce* and *consume* operations required to emulate *decide* and *combine*).

For this example $\alpha = 1$ assembly operations per time step, as before, but the transition firing rate $\tau = 5$. Because of the large number of LFSR steps needed for state permutation and verification, the simulation runs more efficiently using this rate at the cost of granularity in symmetry breaking. The resulting structures would be identical for any transition rate, however. Figures 5.21 to 5.24 illustrate the entire assembly process, from a soup of initially identical units to multiple copies of the target computing device.

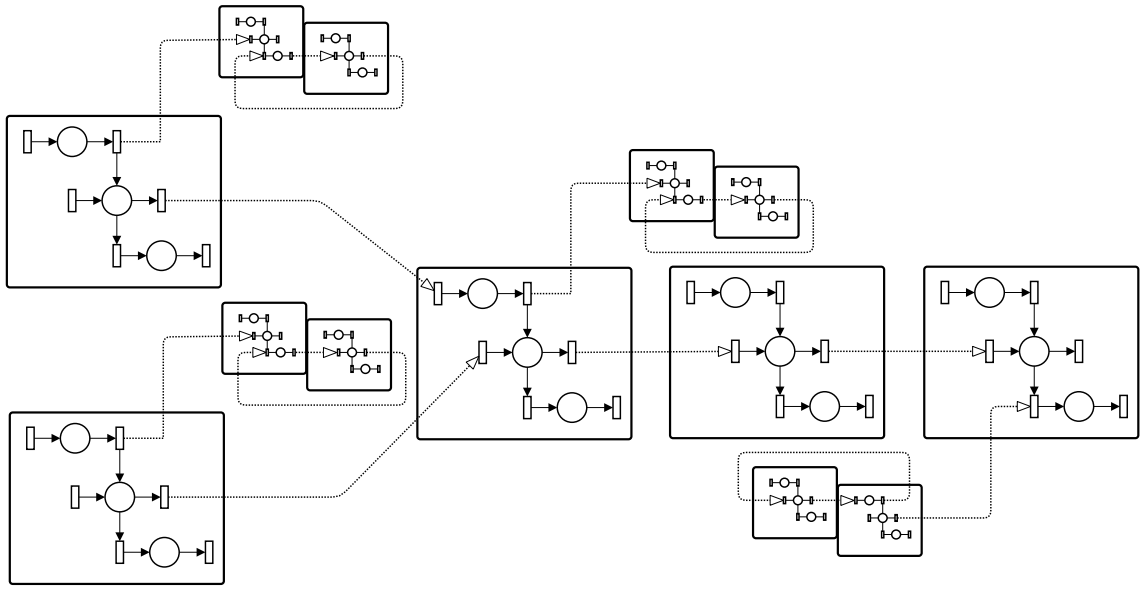


(a) (b)

Figure 5.21: A CORAL simulation containing 256 instances of the C/E unit implementation of Figure 5.19, initially in identical state (5.21a). Controllers are not drawn due to the number of units pictured. After an initial symmetry break to establish seed units, turtle assembly attaches several other units, resulting in the partial tree structures of (5.21b). The target structure is that of Figure 5.20b, and assembly has started generally from the left to right.

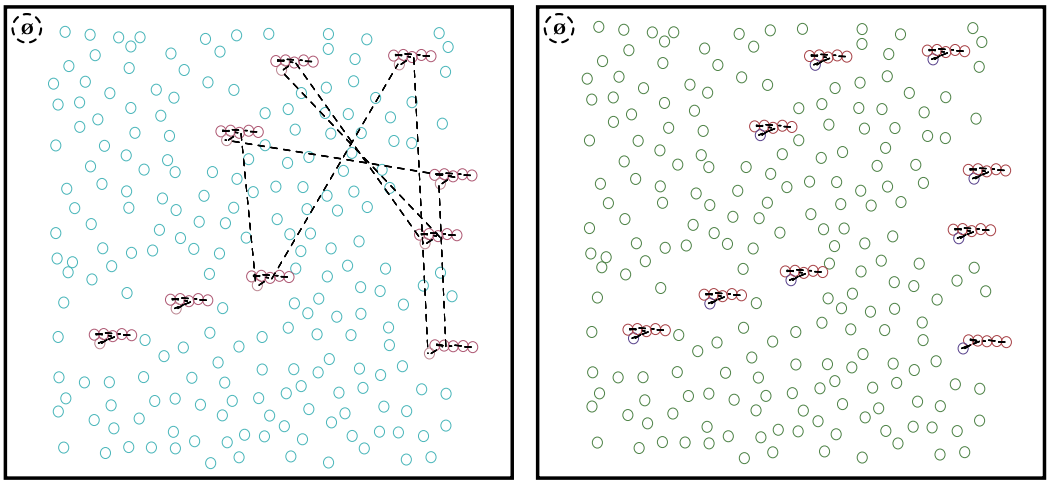


(a)



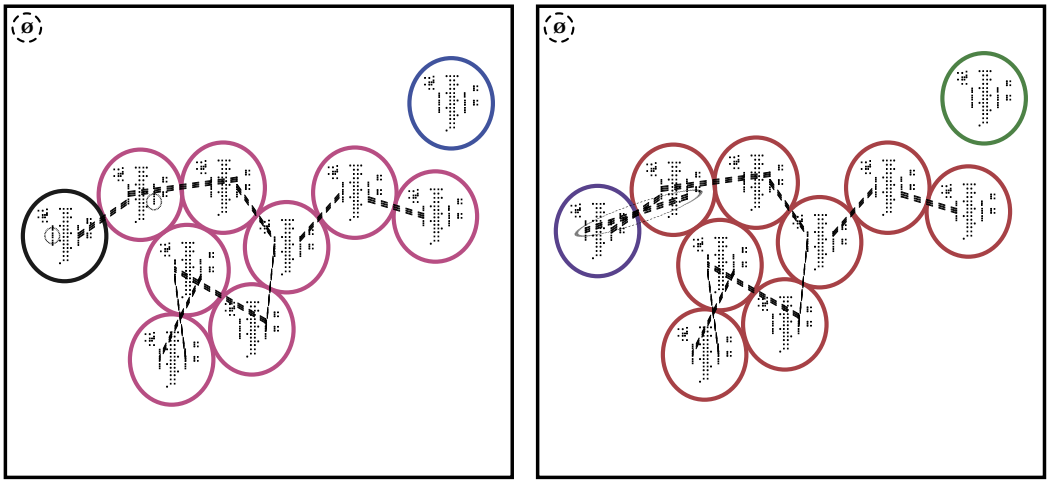
(b)

Figure 5.20: The island graph shown as (5.20a) is the *F* section of Figure 5.8. After the primitive operations are transformed into combinations of the base C/E net primitive (by the identities of Figure 5.2), the localized structure can be built using turtle assembly operations (Figures 5.21 to 5.24).



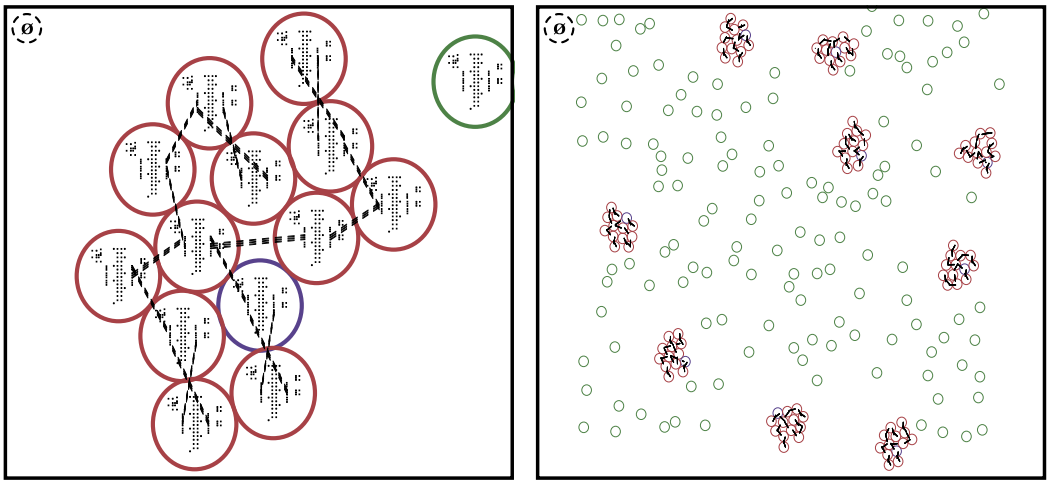
(a) (b)

Figure 5.22: Starting from Figure 5.21b, a new cyclic connection is added between two units (5.22a). This results in some correct self-connections, but also many incorrect connections to other structures. Over several turtle verification unlink/relink steps, non-self-linked structures are unstable and eventually all connections are self-connections.



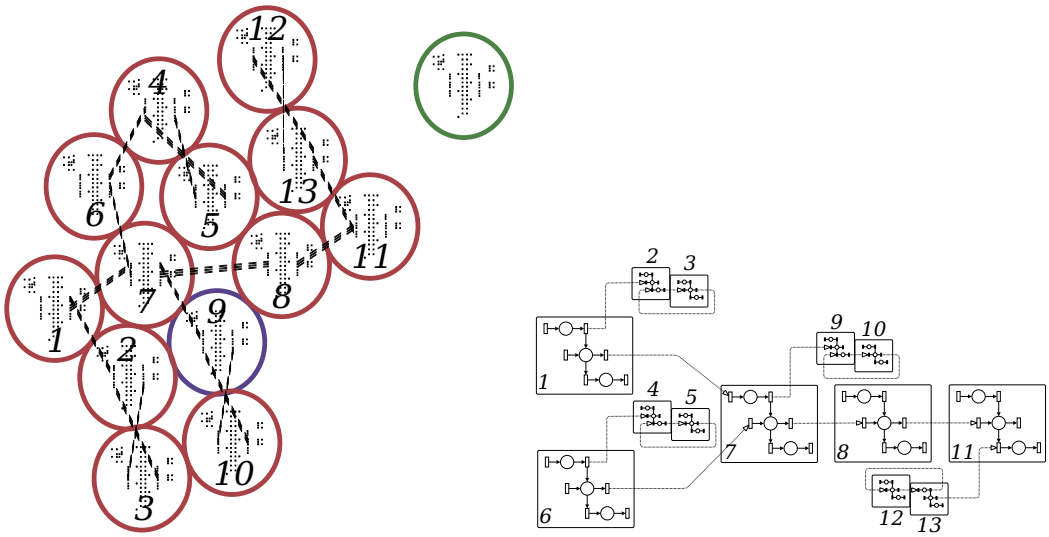
(a) (b)

Figure 5.23: The three *combine* operations of Figure 5.20a have now been partially built, though the topmost C/E primitive structure for the consume operation has not yet been linked back to the structure. The turtle node (differently colored) is therefore at that node and ready to create another cyclic connection back to the structure (ports and connection circled by thinly dashed lines). After several assembly and validation steps, all structures are linked as in (5.23b).



(a) (b)

Figure 5.24: The target structure of Figure 5.20b has now been built fully, as shown in close-up (5.24a). Because of the relaxation algorithm the position of some units are not exact, but the topology of the links is the same (see Figure 5.25). Eight other identical structures are also created, each of which compute identically and can be used as the basis of the assembly of a larger device.



(a) (b)

Figure 5.25: Explicit labeling of a constructed assembly from Figure 5.24, showing that it is in fact an instance of the target computing structure of Figure 5.20b.

5.6 Remarks

This chapter introduces and addresses two limitations of NOR assemblers - the inability to form graph structures and emulate more powerful computing devices. On top of these changes, the formalisms used are simplified to be entirely C/E-net-based, leading to controllable units described in a mathematical language which build more powerful devices in the same language. The C/E unit is the concrete instantiation of these ideas. Assuming an unlimited supply of units, C/E unit

structures can be built which emulate any size C/E net, which in turn means they can emulate a Turing machine with any size tape. This includes building at larger scale the internal structure of the C/E unit itself, which requires many units and much time, but is possible in principle. The assembly manipulations are made more generic and require fewer signals by using parallel state manipulations via LFSR logic. Using this ability, C/E units form and verify graph structures by manipulating special turtle states inside the structures themselves. Essentially, turtle states are the distributed "handles" with which external broadcast signals grasp parts of the unit structures.

The C/E unit is a novel prototype for assembling artifacts with unlimited computational potential while obeying conservation of mass and finite connectivity. Certain modular robotic units with complex internal controllers may be implicitly capable of of complex computations, but only natural systems thus far have used assembly itself as the sole programming step. Chemical computation has also demonstrated computation through assembly such as in (Adleman, 1994; Winfree, 1996; Dittrich & di Fenizio, 2007), though as mentioned in Chapter 2 the physical substrate is pre-specified or pre-tailored for the particular computation. C/E units are assembly amino acids, a general unit which can build any other computational form, and can build any device without prior modification.

The C/E unit is a proof-of-concept that increasingly powerful devices can be built from very limited general parts in a distributed way. The implementation simulated above may seem complex at first glance, partially because of the low-level pieces from which it is built (capacity-1 places and transitions) and the minimal environmental assumptions (no locality and no individual unit interactions). A number of minor simplifications are possible to the knowledge of the author, but any unit will require control over the basic operations of linking, unlinking, enabling multiple ports in different portions of the structure, and verifying cyclic connections. A LFSR is a very general way to achieve distributed state control for all these tasks using a minimum of signals, though perhaps a more integrated control type would lead to a more elegant design.

The formation of graph structures is a particularly challenging aspect of CORAL assembly because stochastic filtering must be used to achieve arbitrary levels of accuracy. As discussed above, no stochastically assembling units, including those of the CORAL model, are able to deterministically create a graph structure with cycles (though with certain environmental assumptions about locality it can become much more likely). These cycles, however, are vital to more powerful computation using either NOR or C/E units. While designing C/E units, one early method considered for validating cycles was to spread a signal from the newly linked units, exploiting the inherent C/E net computation of the partially built structure. Instead of "turtling" through individual units, as described above, a structure might also be "flooded" with tokens from a connected unit. This is practical when the structure is acyclic, but the response of the rest of structure to token input may be that of any arbitrary C/E net once other cycles exist. Determining the output of a general C/E net to general input is a PSPACE-complete problem (and comes ever-closer to non-computable as the size of the structure becomes large); there is no efficient algorithm to determine when or whether a token signal arrives at some other unit without additional restrictions. In consequence, the predictable individual propagation of tokens between pairs of units is used instead. It seems generally difficult to direct further assembly by bootstrapping on the computation of partially-built structures, since the output of computationally powerful structures is difficult, if not impossible,

to predict.

Building meta-C/E net controllers from C/E units (themselves described using C/E nets in the CORAL simulator) is possible to simulate but remains complex. Fully recursive C/E assembly using this idea is not yet possible but is a goal worth pursuing, since it would enable the same types of multi-level control discussed for the less-powerful NOR units. The main challenge is to somehow selectively block certain connected units from responding to particular broadcast signals, as is done via shielding in NOR meta-units. In other words, emulating the C/E logic of the controller itself is not a problem, but there is no way yet to emulate the special signal and port transitions separately from the rest of the units. This limitation does serve to underline the subtle requirements of recursive assembly - even with Turing-complete computational potential and arbitrary state permutations, multi-level control over a system requires the *means of control* itself be emulable.

Little discussion was made so far about the object of this control - what the computation of C/E units can do. No particular physical or chemical environment has been assumed for C/E units to act in, so the description of their function is limited in the CORAL model to the relevant composition and assembly behavior. Given some sort of target environment, the computation C/E structures perform can be linked to structural behavior in any number of ways, and this of course is the primary reason for such computation. To re-use an example from the NOR assemblers, the compression of unit-compressible modules could be linked to particular token constellations of a C/E unit, resulting in different motion from different token flow. The extra expressiveness of C/E nets compared to Boolean functions means that any computable behavior of this type could in theory be built, though the structure of the final assemblies would of course need to be related to the way in which the behavior was implemented. These are important, interesting questions, and the C/E unit is proposed as a prototype for building realistic units which are general computational parts and also potentially general mechanical parts. While not entirely realistic, the C/E unit model does not violate conservation of mass and complementary assembly ports can be built many ways and many sizes.

This chapter concludes the discussion of two designed examples of scalable assembly: NOR and C/E units. In combination they demonstrate that the CORAL simulation is capable of building devices of any size and as powerful as any other computing machine, given simple and general individual parts and a single information broadcast. Nature has performed the same task many times over, but with information encoded in the process of natural selection and evolution. It is possible that novel and simpler assemblers with scalable assembly properties could be discovered using the same types of evolutionary algorithms. In the next chapter, this hypothesis is tested in a series of evolutionary experiments into the automated design of CORAL assemblers.

Chapter 6

Evolutionary Search for Scalable Assemblers

In addition to the designed assembling systems of Chapters 4 and 5, a parallel set of experiments was performed in order to determine whether simple and novel assembling systems could be discovered through evolutionary search. Other experiments evolving group behaviors for robots (Kwong & Jacob, 2003; Baldassarre et al., 2003; Trianni et al., 2003, 2004; Pugh & Martinoli, 2006; Zykov et al., 2007), agents (Koza, 1992), artificial chemistries (Theraulaz & Bonabeau, 1995; Bonabeau et al., 2000) or all three (Studer & Lipson, 2006; Sayama, 2009) have discovered interesting behaviors for groups of agents or robots using a variety of evolutionary algorithms. Virtual evolution has also been applied to realistic chemical simulators, where it is able to generate at least second-order micelle structure (Buchanan et al., 2008). Using the CORAL model as a base, it was hoped that these types of results could be extended to controllable behavior over much larger scales. Evidence toward the lower complexity bounds of assembling units was gained from the results of these evolutionary runs, as well as necessary properties for control to be possible at all. This study is also motivated by the open artificial life question of building a formal framework for *synthesizing* dynamical hierarchies at all scales (Bedau et al., 2000; Lenaerts et al., 2005; Bedau, 2007). The first results evolving pairing assemblers in this chapter were reported earlier as (Studer & Harvey, 2008).

The core methodology of the chapter is the use of evolutionary algorithms (EAs) and other related evolutionary programming (EP) methods to find novel C/E net unit controllers for scalable assembly tasks. Results are partially positive, with the evolution of simple units able to form recursively pairing chains on command. More complex structures were attempted unsuccessfully, but identifying these limitations directly led to addressing them in the designed assemblers mentioned in previous chapters. A sweep using a selection of established and not-so-established evolutionary algorithms and genotype encodings points toward particular methods which may be better suited toward searching this difficult fitness space in the future.

6.1 Evolutionary algorithms and genetic programming

Evolutionary algorithms have become a well-known way of discovering and optimizing potential solutions to simulable problems, particularly when the form of the solution is difficult to engineer

via traditional means. A variety of related research in the 1960s began exploring the evolutionary process as an intelligent search (Bremermann, 1962; Fogel et al., 1965) (summarized in (Fogel, 1998)), which was followed by the development of the so-called genetic algorithm (Holland, 1975, 1992) using a virtual binary chromosome. Many varieties exist today, though a full survey is beyond the scope of this thesis. The term evolutionary algorithm is here defined broadly to mean any algorithm which uses stochastic selection to improve a population of potential solutions. This definition also includes related algorithms like simulated annealing, despite the fact that the inspiration comes from physics rather than biological evolution.

Some of the earliest applications of genetic algorithms were designed to evolve computing models in the form of finite state machines (Fogel et al., 1965). This research has expanded into a variety of active fields, and continues directly to this day ((Keller & Lutz, 2005; Lucas & Reynolds, 2007) have good summaries). The work has also been widely expanded to include the evolution of neural networks (Harvey et al., 1997; Yao, 1999; Nolfi & Parisi, 2002), lisp programs (Koza, 1992), and logical circuits (Koza, 1992; Harvey & Thompson, 1996; Miller et al., 1997; Yao & Higuchi, 1997; Miller & Banzhaf, 2003; Tan et al., 2004; Roggen & Federici, 2004; Koza et al., 2005) among other types of models.

Recently, evolving Petri nets has also become an active research area. Originally the work focused on more traditional uses of Petri nets in job scheduling and workflow management (Chiu & Fu, 1997; Tohme et al., 1999; Saitou et al., 2002), but a new strand of research uses Petri nets to model complex (bio-)chemical systems (Goss & Peccoud, 1998) where Petri nets representing the master chemical equations are evolved (Kitagawa & Iba, 2003; Moore & Hahn, 2004a; Nummela & Julstrom, 2005). Given some set of chemical data or target functions, the object is to create a Petri net model of the system that matches all available data (and is as small as possible).

While superficially this may seem identical to the evolution of Petri nets in the CORAL model (which can be considered a variety of artificial chemistry), the target of the above studies is modeling the full behavior of the entire chemical system, where each chemical structure is represented as a *place* in the full chemical network. In the CORAL model, Petri nets represent *individual* behavior; using the chemical metaphor above, it would be as if the precursor molecules used were evolved to fit fixed interaction mechanisms. As an aside, this use of Petri nets in the CORAL model does not preclude their use as a system model as well, where species of Petri net unit controllers replace chemical species. The idea has been discussed for graph grammar systems (Klavins, 2006), of which Petri nets are a subset, and Petri nets can be elegantly extended to use other Petri nets as tokens.

There has also been limited work in other areas evolving Petri net controllers. An investigation evolving Petri nets as general classifiers is described in (Reid, 1998), which was also a precursor to some of the previous biological work. A novel coevolutionary algorithm is used which rewards each transition of the net individually, similar to the Holland bucket brigade (Holland, 1975). The only robotic example known to the author is (Bourdeaud'huy & Yim, 2002), in which gaits for a hexapod walker are evolved via genetic algorithm.

The basic optimization problem of finding an appropriate Petri net with particular computational behavior is somewhat similar to the previous job scheduling, chemical, and robotic examples, with a significant difference in the type of function required. Instead of matching a single

behavior or set of data, a multi-level behavioral test is required. The work below represents a first application of Petri net evolution to distributed assembly or multi-level behavioral tasks.

6.2 Pairing assemblers

The scalable assembly behavior investigated using evolution was designed to be as primitive as possible, so that evolution would have the best chance of discovering solutions. As discussed in Chapter 3, the basic assembly operation of units in the CORAL simulation is to pair with one another, and it is possible to generalize this pairing behavior to assemblies of multiple units. While for a single unit, such pairing behavior is almost trivial to implement (enable a port transition on *either* one side or its complement), the same behavior for many linked units requires synchronization of open ports at the ends of the large chain structure. A single signal indicates that the next pairing operation is meant to take place; coevolution of the signal and controller is not investigated. Figure 6.1 illustrates the idea.

This recursive pairing behavior requires only two ports, and so only a single pair of complementary ports are used for each atomic unit. With only two complementary ports, the "sea" of components in a simulation instance will generate only pair, chain, and loop structures, whatever the structure size, as seen from the example assembler simulated in Section 3.7.1 with two ports. As was also seen in Figure 3.21, without a properly designed controller CORAL units generally reach an uninteresting steady state in which the structures built are immune to further perturbation. The goal of the experiments in CORAL unit evolution is to discover assemblers which do *not* become immune, and respond to background signal input in a controllable way *no matter how many recursive pairings have occurred*.

6.3 Evolving recursive assemblers

A standard evolutionary algorithm, the Microbial GA (MGA) (Harvey, 2001, 2009), was used to evolve the C/E net controller placed on the many identical atomic units in a CORAL simulation. As in a traditional genetic algorithm (GA), a MGA maintains a population of genotypes which are ranked, selected and filtered via a fitness function. However, MGAs use a steady-state population, where genes from selected genotypes simply overwrite the genes of less preferred genotypes, proportional to a constant value ρ. In practice, this considerably simplifies the operation of a Microbial GA (as compared to a generational GA which updates many units at once) while maintaining similar performance in many search domains. Results from other types of evolutionary algorithms are also discussed later in the chapter.

6.3.1 Genotype representation

For performance and simplicity, binary genotypes were used to evolve the unit controller C/E nets. For p numbered places and t numbered transitions, every possible outgoing edge from a place to a transition is assigned a single bit: 1 indicating the edge is present in the controller, 0 indicating the edge is absent. Effectively, the adjacency matrix of the C/E net is directly represented as a binary string.

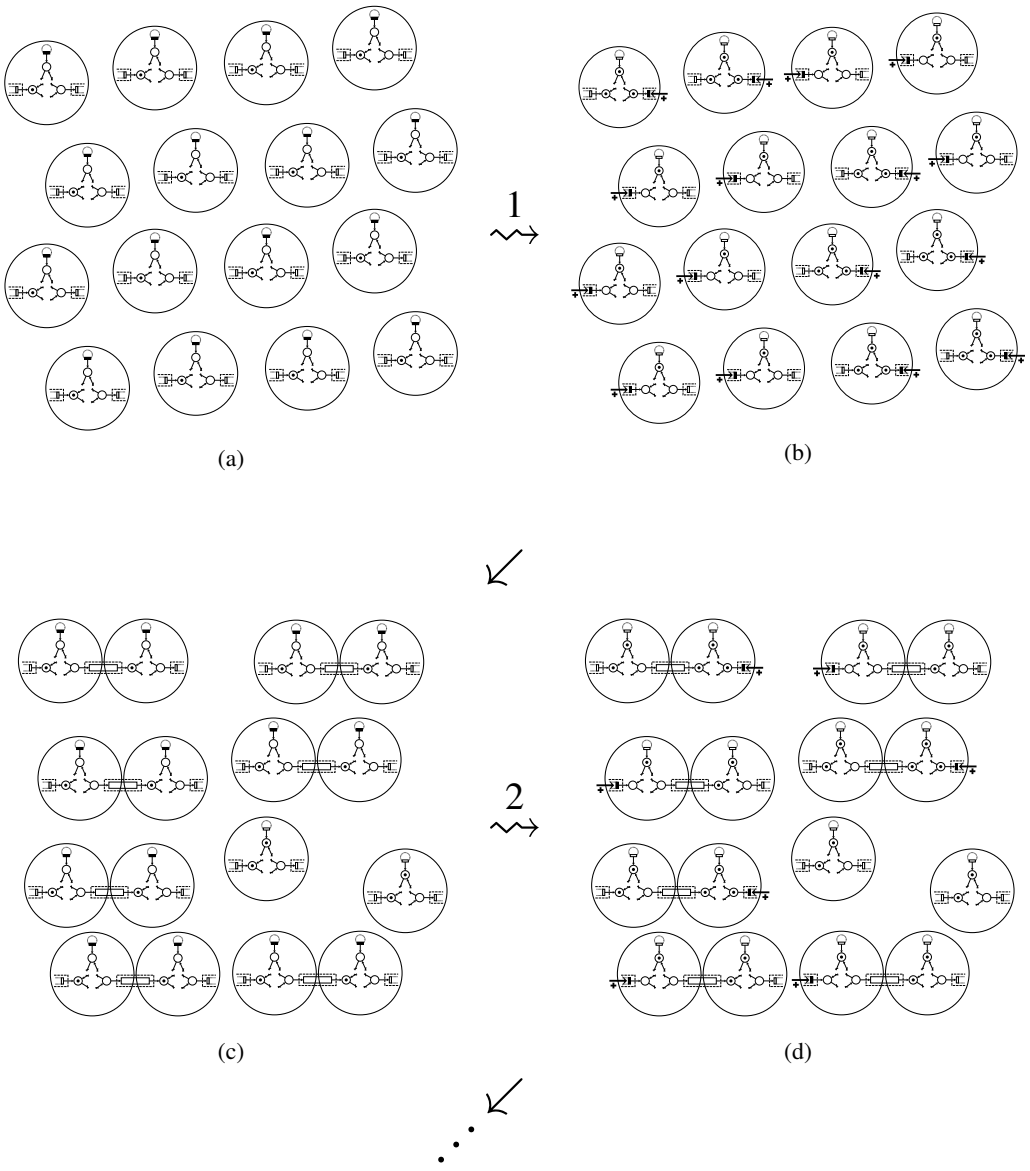


Figure 6.1: Recursive pairing of CORAL units. In (6.1a), all units are identical and contain some undefined controller with two ports and a signal transition. After a signal is sent to the units (indicated by the $\rightsquigarrow$ arrow) the units differentiate into two types, enabling either the left or right port but not both (6.1b). Once assembly occurs (6.1c), the correctly paired units again are ready to receive a second signal. In response to the second signal, each pair opens either the leftmost or rightmost port (6.1d). Assembly occurs again, creating groups of four. Ideally the process can be repeated indefinitely, for unlimited numbers of pairing signals.

A significant factor allowing this representation (which could not be used in many of the biochemical evolutionary Petri net studies) is the lack of weighted edges in C/E nets. ((Nummela & Julstrom, 2005) find weights by alternative methods, though the topologies are limited.) Instead, the biochemical studies used representations of lists of tuples which encode $(place, weight)$ pairs for each transition, as does the robotic work of (Bourdeaud'huy & Yim, 2002). Effectively, though there are minor differences in format between studies, the tuple representation evolves an adjacency matrix encoded in sparse form. The grammatical evolution approach of (Moore & Hahn, 2004a) is a notable exception, and generates weighted networks via the application of a custom context-free grammars. While this approach seems quite interesting, because evolved controllers for scalable assembly had not been investigated previously it was decided that the simplest encoding which did not require assumptions about good result topologies would initially provide the clearest results. The studies evolving assembly-line automation control assume predefined architectures, so the encodings are not general and cannot be used in this context.

Each place is also assigned a single bit, indicating the presence or absence of a token at that place in the controllers' initial marking. Using this format, $(2pt + p)$ bits are able to fully specify any CORAL controller with p and t transitions. The transition assigned *a priori* to receive background signals and the two transitions required to link and pass tokens at unit ports are then *added* to the t transitions with evolved links and attached to the first three places of the evolved C/E network. Only a single transition per port is needed for both linking and communication, because link transitions also function as synchronized transitions after a connection is made. Unlink transitions are not required for recursive pairing behavior in the evolutionary search, and were not included in the experiment.

Figure 6.2 below is an illustration of the evolved CORAL unit. The ports are labeled the standard *In* and *Out*, and a single signal transition for *G* is defined, which is the minimum required for controllable assembly. CORAL assemblers with more port transitions and signals were found to be difficult to evolve, and later work in this chapter comparing different encodings and and evolutionary algorithms begins to address this issue.

6.3.2 Fitness function

To evolve units which maintain a pairing assembly ability as they grow into larger chains, a fitness function is required which evaluates the pairing behavior of units at any scale. The core idea is to measure the number of structures of the correct size after a single "level" of assembly. If pairing occurs, the structure sizes will be correct. The converse, of course, is not always true, since there are other behaviors which generate structures of the correct size at higher pairing levels by assembling mismatched parts; e.g. structures of size 3 and 5 or 2 and 6 forming structures of size 8. These mismatched parts necessarily incur a fitness penalty at lower assembly levels, so there is always evolutionary pressure to reduce the number of these units toward the maximum fitness at which pairing always occurs. Counting the number of units of particular sizes is fast, simple, and as discussed above gives a gradient toward the ideal solution, and so was considered appropriate for the scaling assembly problem.

The recursive pairing behavior evolved is also required to be minimally controllable in order to be well-defined. If pairing always happens spontaneously, without the need for a background

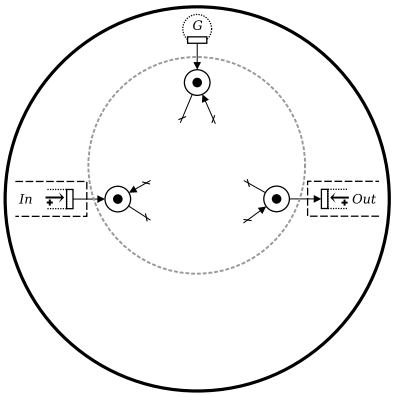


Figure 6.2: The skeleton places and environmental transitions of the evolved CORAL unit. *In* and *Out*-port transitions and the input transition for the *G* signal are linked directly to place nodes. Other internal transitions and places of the controller are not linked to port or signal transitions. This is indicated by the gray dotted circle, outside of which no C/E net controller edges are evolved.

signal, it is unclear where one should divide behavior for each scale. Spontaneous pairing would be a perfectly acceptable mechanism for assembly of long strings of units, however this study focuses on evolving an assembly response to particular kinds of input, *controllable assembly*, which is preserved across all scales. The input is designed to be as simple as possible, consisting of setting, before each recursive pairing behavior or *level*, a *G* background signal (*Go*-signal) after 200 CORAL environment time steps. Compared to the designed assemblers this is quite a simple input, but the recursive pairing task does not require complex structure to be specified via the background signals. Each pairing level requires only a new single edge, and so a single signal was hypothesized as sufficient.

The full fitness function is slightly more complex than the previous description, and is presented in all relevant detail as Algorithm 3. The main addition is a fast "sanity check" using a small simulation environment to ensure units do any sort of assembly at all and do not spontaneously assemble. First, the genome is translated into a C/E net controller using the encoding from Section 6.3.1 and the small and large CORAL simulation environments of many units are initialized. The controller is first tested in the small environment to ensure that it does not spontaneously start assembling anything without input and that it does assemble something with *G* input. If this is not the case, the fitness evaluation terminates and adds a fractional value designed to bias toward units with many transition firings over inactive units. If the small tests succeed, the pairing tests are performed with *G* input in the larger simulation environment with many more units. For the first set of tests, the larger environment is reset with the initial controller phenotype, but unlike the small simulation the large simulation environment is not reset on subsequent tests. The large environment, over multiple assembly levels, retains all the incorrectly paired units of the previous levels and simply has a *G* background signal set for each assembly interval.

After the signal and 200000 time steps, the number of structures of the correct sizes are counted and the fitness value updated. If pairing seems to have occurred, the same tests begin again, searching for pairs of size $2^{level+1}$ (where $level = 1$ for the first pairs). When less than two pairs

Algorithm 3 Recursive pairing behavior fitness function in Python pseudocode.

```
# Initialize the sim environments
input_step = 200; G = ...
large_sim = init_sim(1024, input_step)
small_sim = init_sim(128, input_step)

def fitness_function(genome):

    # Translate the controller from the genotype
    controller = to_phenotype(genome)

    # Test fitness at each level
    fitness = 0; level = 1; pairs = [controller, controller]
    while len(pairs) >= 2 and level <= 10:
        num_conns = test_sims(pairs, level)

        # If sanity tests have failed, add fractional activity fitness
        if num_conns == None:
            fitness += last_sim_activity() / 1000000
            break

        # Get pairs and add to fitness
        pairs = get_structs(large_sim, 2^level)
        fitness += len(pairs) * 2^level * level^2
    return fitness

def test_sims(structs, level):

    # Do a small quiet/noisy pairing test (if structures fit)
    if 2^level <= 2 * len(small_sim.units):
        steps = get_small_sim_steps(level)
        not_sane = (test_sim(small_sim, structs, [], steps) != 0
          or test_sim(small_sim, structs, [None, G], steps) == 0)
        if not_sane: return None

    # Do a pairing test in the large sim, resetting only initially
    return test_sim(large_sim, level == 1 ? structs : None,
                    [None, G], 1000 * input_step)

def test_sim(sim, structs, input, steps):

    if structs != None: reset(sim, structs)
    sim.set_input(input); num_conns_made = sim.step(steps)
    return num_conns_made

def init(sim): ...
# Resets the simulation environment using the provided structures
def reset(sim, sample_structs): ...
# Returns the structures in the simulation, potentially of a given size
def get_structs(sim, size=None): ...
# Gets an appropriate number of small simulation steps
def get_small_sim_steps(level): ...
# Returns the number of transition firings in the last sim test
def last_sim_activity(): ...
```

are found, indicating the next level of pairing must be unsuccessful, the tests end. The full formula for fitness of a particular genome g which generates a number of pairs n_l at each level l is:

$$fitness(g) = \sum_{l=1}^{\infty} (n_l \times 2^l) \times l^2 \tag{6.1}$$

The above is a fairly straightforward summation using the number of pairs times a factor for each level. The product $(n_l \times 2^l)$ is just the number of units paired correctly at each level, while the extra l^2 factor weights the pairings progressively greater as the units grow, which biases the evolution toward units which work partially at multiple assembly levels and against units that work better at a lower assembly level. Fitness tests were stopped in practice after 10 levels of pairing, since there are initially $2^{10} = 1024$ units in the large simulation.

6.3.3 Pairing results

The parameters used in the MGA algorithm and CORAL simulation for the evolution of pairing units are:

MGA Algorithm:

- Population Size: 100 genomes
- Mutation rate: 0.01 mutated bits per crossover
- Uniform crossover with probability $\rho = 0.5$
- Tournament selection
- Binary encoding of $(2pt + p)$ bits (signal and link transitions defined *a priori*)

Simulation parameters:

- Transition activation time $\tau = 50$ timesteps
- Average assembly rate of $\alpha = 10.24$ assembly operations per timestep (an earlier iteration of the CORAL framework was used in which assembly rate is proportional to the number of units)
- 200000 time step wait until the large assembly is considered finished
- Input string of :200,*G*,:199800 at each level

For Petri nets of sizes 3 places and 6 transitions (3p/6t), 5 places and 8 transitions (5p/8t), 7p/10t, 9p/12t, 11p/14t, 15 evolutionary runs of the MGA were performed. The different sizes were chosen in order to gather evidence for a minimum complexity threshold at which scalable pairing behavior emerged. The two necessary link transitions for the *In* and *Out* ports and the *G* signal transition are always present, which accounts for the imbalance of three transitions, along with incoming and outgoing edges to these transitions from three of the inner places. Other unreported tests initially indicated that the evolution of consistent pairing behavior becomes much harder without these edges, though results presented later relaxes these assumptions.

In each trial the fitness values of the evolved populations were tracked on 200000 fitness evaluations (this corresponds to approximately 2000 generations of a generational GA). Where runs discovered better portions of the fitness space, the fitness function could slow quite dramatically

	3p / 6t	5p / 8t	7p / 10t	9p / 12t	11p / 14t
Genome Size (bits)	21	55	105	171	253
Max Fitness	24687	129559	222567	117463	79031
Max Pair Size Created (units)	128	256	512	256	128
Max Assembly Levels Achieved (Std. Dev.)	7 (0.99)	8 (1.36)	9 (1.44)	8 (1.46)	7 (1.85)

Table 6.1: Results from each set of 15 pairing evolutionary runs, over 200000 fitness evaluations. The best assembler was found using 7 places and 10 transitions, though the result could be simplified to a 5 place and 8 transition network. As the complexity of the network grows, better solutions tend to become harder to find, indicated by higher variance in assembly levels between runs. The minimal fitness contribution of the activity penalty is also truncated in these results.

and many fewer evaluations were possible. Evolutionary runs were ended after approximately two weeks of computation time on a shared computer cluster, if not finishing sooner. Results from these are summarized in Table 6.1 and Figure 6.3. Assembly level units indicate how many recursive pairings were performed before no further pairings were detected. For example, an evolved controller that when tested in simulation produces pairs in response to the first *G* signal, but these pair structures fail to function further in response to further *G* signals, reaches an assembly level of 1. A different C/E controller able to form pairs, pairs-of-pairs, and pairs-of-pairs-of-pairs (creating units of size 8) reaches an assembly level of 3. The ideal result is to find unit controllers which, after pairing, always function as a larger pairing unit, giving a theoretically infinite assembly level. In practice only 10 recursive pairing events were tested (for a max fitness of 10) because of limitations in the number of units simulated and the stochastic symmetry breaking required to form pairs.

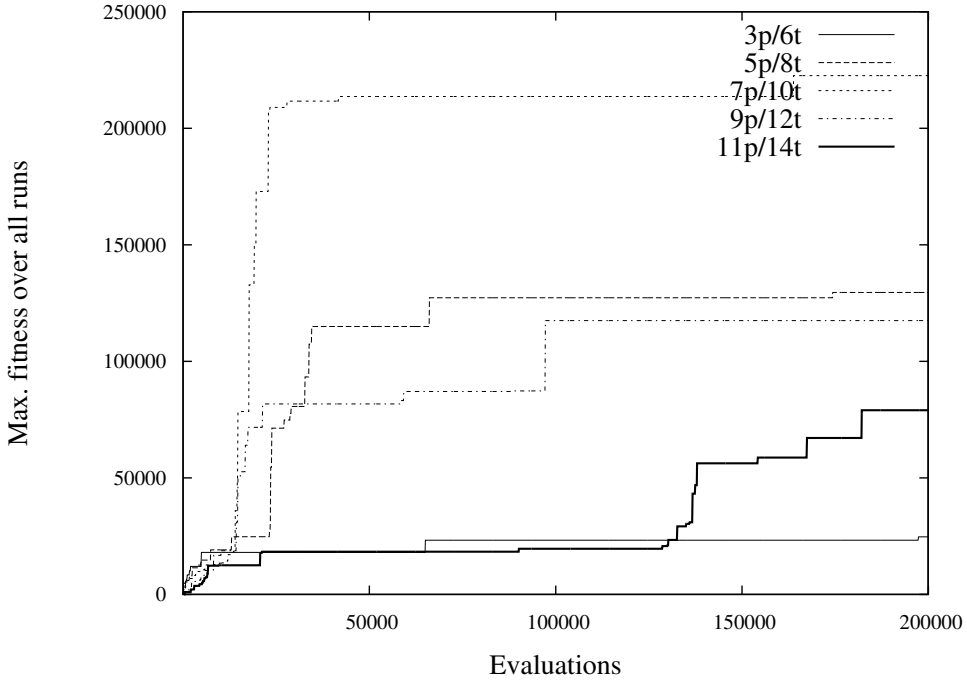
The full source code for the above tests is available online at:

- https://coralassembly.wordpress.com/

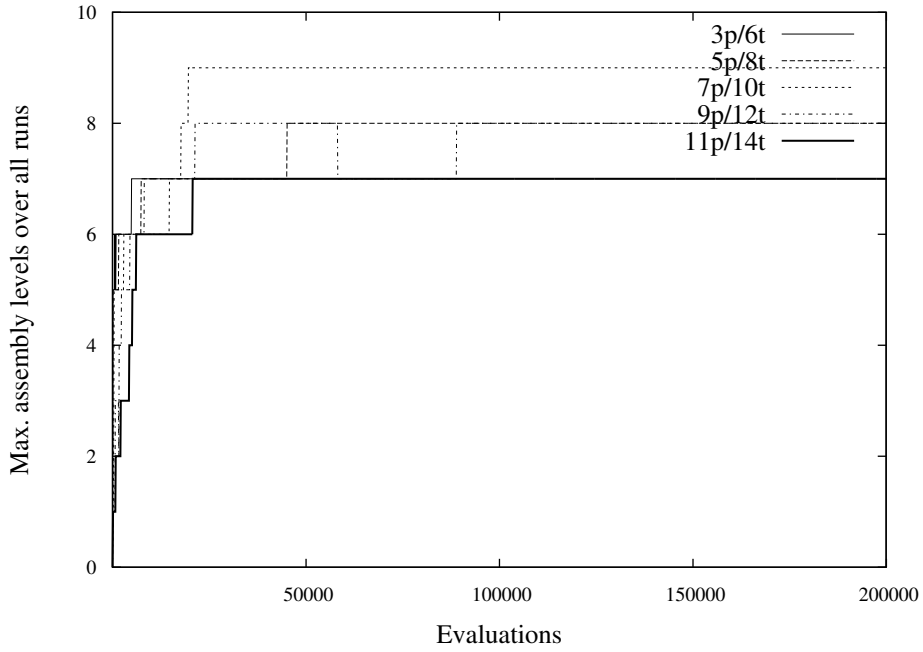
A slightly modified version of the Petri Net Kernel (PNK) framework (Kindler & Weber, 2001) is included in order to support loading and saving to PNML (Petri net markup-language) files. An optimized portion of the Attributed Graph Grammar (AGG) framework (Rudolf, 1997; Melamed, 1998) is also used in the transformation from genome to C/E net. The software for each can currently be found online at:

- Petri Net Kernel (v2.2): http://www2.informatik.hu-berlin.de/top/pnk/
- AGG framework (v1.6.1): http://user.cs.tu-berlin.de/∼gragra/agg/

As can be seen from the results in Table 6.1, the maximum assembly levels achieved grows with the addition of places and transitions until 7p/10t is reached, at which point the maximum assembly levels drop back again. In rare tests, random assemblers can form at least one structure of the correct size for many levels through lucky choices in environmental pairing, but this effect becomes much less likely for 8, 9, or 10 pairings, since the number of pairs possible decreases

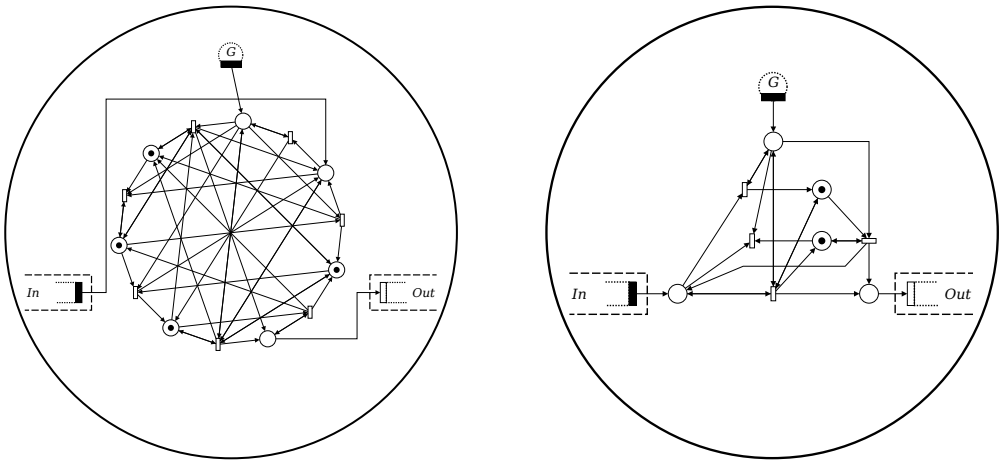


(a)



(b)

Figure 6.3: Maximum fitness (6.3a) and assembly levels (6.3b) from each set of 15 pairing evolutionary runs, over 200000 fitness evaluations. As reported in Table 6.1, the best assembling controller was found using 7p/10t genomes. While fitness increases continually, there is a temporary decrease in assembly level of the 9p/12t assembler between 50000 and 100000 evaluations due to more pairs at lower levels.



(a) (b)

Figure 6.4: Best final evolved assembler, from the (7p/10t) evolutionary run (6.4a). Extra places and transitions which do not fire or fire initially only once were removed to create the simplified version of (6.4b).

significantly. The assemblers discovered of size 3p/6t were all essentially random assemblers exploiting this fact.

There appears to be a minimum level of C/E net complexity (in terms of places and internal transitions) below which the MGA is not able to discover good recursively pairing solutions. Controllers with more transitions and places than this cutoff are, in theory, capable of least this performance or better. In our experiments, it was found that a finite, rather small amount of evolved C/E logic (5p/8t) is able to perform the optimal scaling behavior. This C/E network was strangely discovered as an evolved 7p/10t unit, which was simplified by removing inactive places and transitions. The controller behaved by opening either the extreme left or right port of each assembled structure in response to a *G* signal, where the port was chosen via symmetry breaking. Assuming the number of units is large, this behavior results in recursive pairing behavior with an arbitrarily small chance of unpaired units at each step.

C/E controllers evolved with only 3 places and 6 transitions were not found to be able to duplicate the behavior of the other, more complex evolved units and therefore only achieved recursive pairing via lucky choices in random pairing. As described above, the fitness function does not check directly for pairing behavior, only structure size, so significant fitness is possible with this random behavior. C/E nets evolved with more places and transitions (9p/12t, 11p/14t) were found that paired more effectively than the very simple (and random) 3p/6t C/E nets, though required many more evaluations than the 7p/10t and 5p/8t nets. There are only 2^{21} genomes for the 3p/6t assemblers (edges to environmental transitions are not evolved), which is a rather small search space. Even in this much more limited area evolution was unable to find correctly pairing controllers, with a smaller assembly level variance than all the other evolutionary runs. Limited numbers of evolutionary runs were also extended for 9p/12t and 11p/14t genomes, which demonstrated slow performance increases seeming to plateau at 8 assembly levels.

Best Pairing Controller

The best pairing unit, taken from the 7p/10t set of evolutionary runs, is shown as Figure 6.4. A simplified version with computationally unneeded transitions and places removed is also shown. The Petri net analysis technique called *unfolding* was used to find redundant places and transitions given the background signal (McMillan, 1995; Esparza et al., 2002), which generates a finite partial order of firings. Unfoldings transform Petri net execution into weaved graph of branching actions, which is also useful for interpreting how evolved Petri nets work. The transitions which never fire and places which never change can then be removed using this graph of actions or alternately by using a full state expansion.

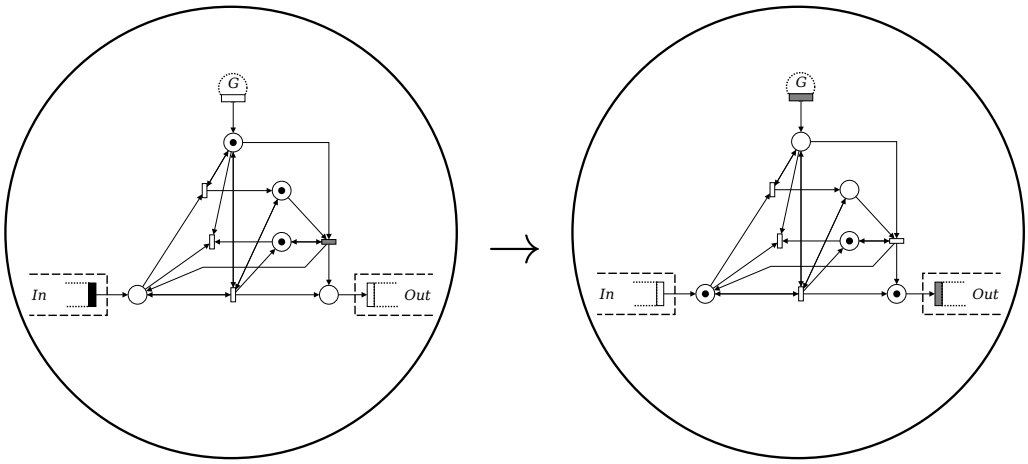
When many of these units are placed into a CORAL simulation environment and a *G* signal is sent, the units form sets of pairs (with a small number of units unable to pair). After enough time steps have elapsed for assembly and for signal transitions to hold, a second *G* signal can be sent as a background signal resulting in groups of four, and so on until the units are exhausted. Assuming the pairing is complete after each G signal (which is unlikely when simulated), these units are capable of pairing themselves indefinitely into larger and larger structures. For any individual test, however, leftover units tend to eventually result in incorrect pairs that finally overwhelm the entire simulation. These leftover units can, in theory, be avoided if ports are opened in an alternating fashion, but this solution was not discovered in any of the evolutionary runs, and may require C/E nets more complex than the maximum of 11p/14t and/or more synchronized port transitions.

Intuitively, the best evolved unit logic works in one of three modes: *initial*, *pass-through* and *decide*. Units start in *initial* mode, with an open *In*-port. After receiving the first *G* signal, the unit may change into an *Out*-port enabled unit (Figure 6.5). This transition does not happen simultaneously for all units, and so some units enable their *Out*-port link transition while other units are still *In*-port enabled. Complementary units may then be paired, usually resulting in most or all of the units ending up in pairs given the assembly rate chosen. Pairing using a default-enabled *In*-port is a special case for the first pairing step, and all later recursive pairings use *decide* and *pass-through* modes to open ports after a signal. After the initial pairing, connected C/E controllers settle into one of these two different states because of the non-symmetric token passing (Figure 6.6).

In *decide* mode, a connected unit "chooses" to enable either the open *In*-port link transition or the connected *Out*-port link transition in response to a *G* signal. In *pass-through* mode, the unit simply passes a token from its *In*-port link transition to enable the *Out*-port link transition in response to a *G* signal. As the unit pairs grow into long chains, the unit at the open *In*-port end of the chain always *decides* whether the chain will enable the extreme *In*-port or *Out*-port, and the other units in *pass-through* mode send *Out*-port tokens onward. This creates chains with pairing behavior like individual units in response to *G* signals, opening *either* the *In*-port or *Out*-port, and can form pairs of chains no matter how long the chains grow (Figure 6.7).

6.4 Fitness landscapes of scalable assembly

Limitations to the results and methodology presented above became apparent after these and other evolutionary results were gathered. Even given the simplifying assumptions of the single signal and limited topology of edges to signal and link transitions, the evolution of assembling units

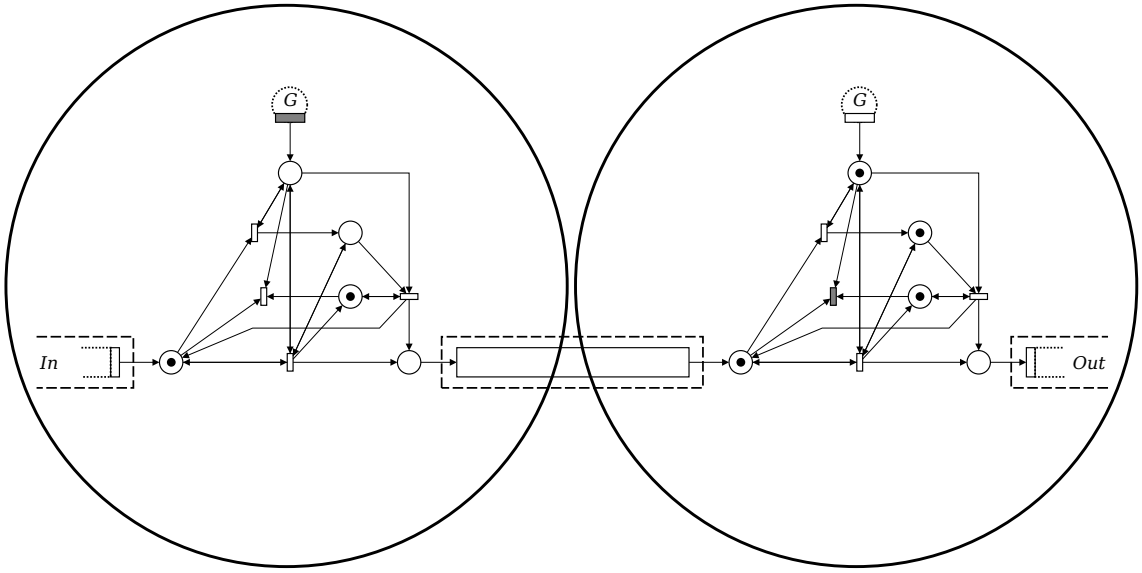


(a) (b)

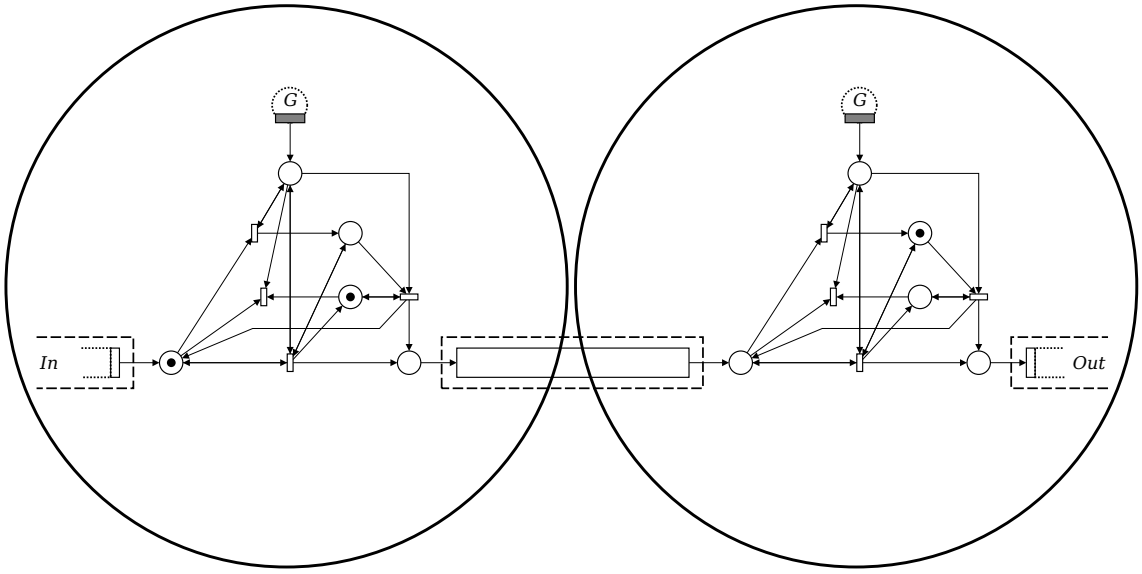
Figure 6.5: The effect of a *G* background signal on the evolved assembler of Figure 6.4, in the *initial* state. When input is received, a controller transition may fire which switches the activation of link transitions. If the transition does not fire immediately, the unit of (6.5a) will be linked at its *In*-port to another unit whose transition fired more at an earlier time step (6.5b). The resulting paired structure is shown as Figure 6.6.

was extremely time-consuming with high variation between runs. Because better assemblers must be tested over longer simulation time, populations which evolve to contain many good assemblers take many times as long to evaluate. Worse, good and bad assemblers can contain C/E net portions which fire continually but have little or no effect on assembly, again causing major slowdowns to the simulation framework. These slowdowns made it difficult to initially experiment using multiple evaluations per unit, despite the stochasticity inherent in the simulation. In addition, often extremely long time periods pass without improvement, only to be followed by large unexpected jumps of fitness. For example, though only more limited runs extended past 200000 evaluations, the 5p/8t and 9p/12t extended runs would sometimes, suddenly, discover assemblers as good as the best 7p/10t evolved assembler of Figure 6.4 (though never for 3p/6t or 11p/14t evolved units).

These problems are well-known issues for discrete and computational evolving systems. Sudden jumps in fitness are general characteristics of rugged (and neutral) fitness landscapes (Kauffman, 1993; Barnett, 2001). Researchers in evolutionary robotics and evolvable hardware have long noted that behavioral evaluation is complex and often subject to high variation (Mataric & Cliff, 1996; Harvey et al., 1997; Yao & Higuchi, 1997; Tan et al., 2004; Nelson et al., 2009). Optimizing multiple, nested types of behavior using evolutionary algorithms remains a difficult unsolved problem in the general case. Behavior shaping (Dorigo & Colombetti, 1994; Saksida et al., 1997) or hierarchical reinforcement learning (Barto & Mahadevan, 2003) or chaining (Bongard, 2008, 2009) address these issues partially, though higher-level descriptions generally require human-assisted division of tasks. The fitness function above uses similar ideas in the small pretest simulation environment and the increasingly large pairing tasks. Unfortunately, in our tests, compositionality puts a natural limit on the order in which the tests can be presented and it is not clear that smaller assembly tasks are "simpler" in any sense than assembly of composed structures. In the interesting cases searched for, in fact, the overall assembly behavior should be the



(a)



(b)

Figure 6.6: Units which have paired from the symmetry-breaking initial state (6.6a) are differentiated and settle into two different modes once assembled. The *decide* mode, set in the left unit of (6.6b), enables either the open *In*-port or the connected *Out*-port link transition in response to a *G* signal, while the *pass-through* mode set in the right unit of (6.6b) will send a token from the connected *In*-port to the open *Out*-port.

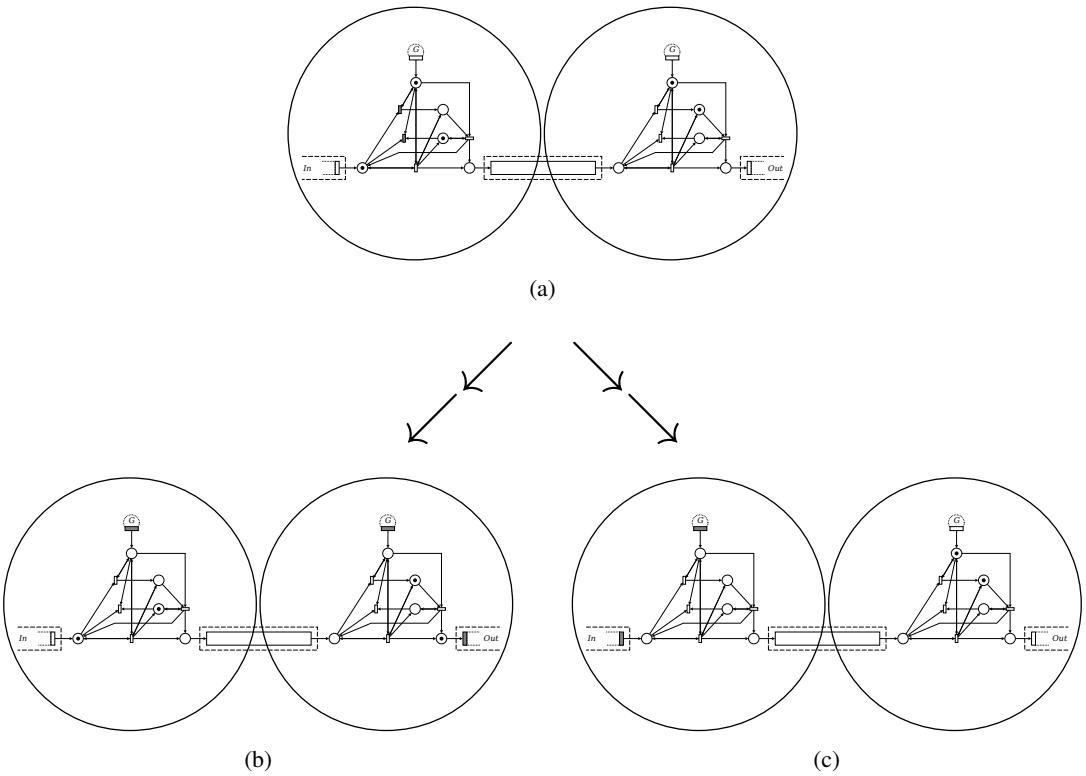


Figure 6.7: After receiving a second *G* input signal, the paired units of Figure 6.6 are able to open either the leftmost *In*-port or rightmost *Out*-port, depending on the stochastic transition firing in the left *decide*-mode unit 6.7a. Depending on the particular transition firing first, the pair structure either enables the extreme *Out*-port (6.7b) or *In*-port (6.7c) link transition. After assembly with another pair, the new size-4 structure will also consist of a single leftmost unit in *decide* mode with all other units in *pass-through* mode.

same between levels. It is possible, however, to upgrade the small/large simulation distinction with finer granularity and to slowly increase the size of a simulation to allow for better (and unlimited) assembly simulation.

In this section, the structure of fitness landscapes for the recursive pairing task are investigated as a study into improvements needed for the evolution of scalable assemblers. A new, slightly generalized fitness landscape of recursive pairing assemblers of low dimension can be imaged directly, while statistical measures are derived for higher-dimensional landscapes. A new fitness function integrating less granular scaling and pair counts is first described below. Since several iterations of the evolutionary framework elapsed between this and the previous study, the fitness landscape of the original fitness function of Algorithm 3 was not probed directly. The modifications largely generalize the previous fitness function in a simpler way, however, and it is believed that the structure of the landscapes is qualitatively quite similar. (Adjustments while developing the newer fitness function tended to preserve the overall structure.) Results comparing a variety of evolutionary algorithms and genotype encodings using this new fitness function are also presented later in this chapter. Several find pairing solutions to the unconstrained pairing assembler problem similar to the original pairing assembler, but a non-population-based method tends to be faster and better at finding good solutions.

6.4.1 Generalized controller and fitness function

Evolvable edges to environmental transitions

As mentioned above, the evolved pairing assemblers of Section 6.2 are limited in that the connections of port and signal transitions to the evolved portions of the controller are fixed. The restriction is lifted in the new fitness function. Instead of the evolved portion of Figure 6.2, edges to and from the signal and link transitions are included in the evolutionary set (Figure 6.8). The set of controllers which can be evolved using this scheme form a superset of the previously evolved controllers, given a particular number of transitions and places.

Scaling fitness

Because of the difficulties evolving hierarchical behavior and the time-intensive operation of the previous fitness function, the new fitness function was also designed to take advantage of a slowly-scaling simulation with more flexible pairing criteria. Unlike the previous fitness function which evaluates a simulated assembler until no more pairs are theoretically possible, the termination criteria here is changed to detect when the target structure sizes of a particular assembly level match poorly with the actual sizes. A similarity measure σ of structure sizes x and y is defined as follows:

$$\sigma(x,y) = 1/(1+|x-y|)$$

The function varies smoothly between 0 and 1, with a peak when $x = y$ and a minimum when $|x-y| \to \infty$. An example helps to illustrate: given a target size of 8 units per structure, structures of size 5, 6, 7, 9, or 10 have a σ of 1/4, 1/3, 1/2, 1/2, or 1/3, respectively. The average similarity value over all structures with size > 1 in the simulation environment can then be calculated using the σ measure. If this average is > 0.6, which ensures that many units must have paired perfectly (otherwise the average cannot rise above 1/2), the next recursive pairing task is tested. Essentially

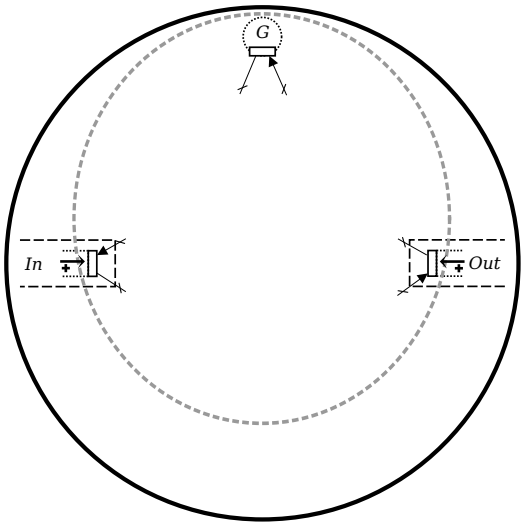


Figure 6.8: The new evolved CORAL skeleton. The *In* and *Out*-port transitions and input transition for the *G* signal are still fixed, but the internal places and transitions and edges between them may now have any form.

the above fitness function smoothly scales the fitness of structures which do not match the pairing size exactly. The scaling can be made more aggressive by multiplying $|x-y|$ by a large factor; at the extreme case the similarity essentially becomes the previous count of pairs.

The sanity check of a quiet fitness test (where no input is sent to ensure units are not pairing spontaneously) has been eliminated in the new fitness function. It was found in the previously-reported experiments that units are unlikely to spontaneously pair past the first pairing level at a rate which matches the input signals. A fast state exploration is instead used to filter out controllers which will never fire link and/or sensor transitions assuming no environmental holding restrictions (as previously mentioned in Section 3.6). If environmental transitions do not fire the units would not be meaningfully evaluated in any case, since without enabled link transitions assembly does not occur and without sensor input the units are again unlikely to pair spontaneously at a rate slow enough to result in more than one recursive pairing evaluation.

The full fitness function, generalized and simplified from that of Algorithm 3, is presented as Algorithm 4. Other small differences include the input syntax "\${*r*-*s*}" indicating a randomized duration of time between r and s timesteps and a reduced number of total timesteps per level. Randomizing time prior to presenting G ensures evolved results are insensitive to the delay. More significantly, fitness is averaged over $n = 10$ independent evaluations, weighted by the fitness value ordering. If these fitnesses are sorted from lowest to highest such that the fitness of the lowest is f_1, the weighted average fitness μ_{wt} is then:

$$\mu_{wt} = \frac{\sum_{i=1}^{n} f_i \times i}{(n^2+n)/2}$$

Essentially the higher fitness values are counted more times than the lower fitness values. This formula is intended to reward units which, despite a few bad pairings due to the stochastic nature of the simulation, are otherwise able to pair consistently.

The weighted average fitness value of the previously-evolved best recursive pairing assembler

Algorithm 4 Simplified and optimized recursive pairing behavior fitness function in Python pseudocode.

```
# Constants
pairs_per_level = 10

def fitness_function(genome):

    # Translate the controller from the genotype
    controller = to_phenotype(genome)
    if not is_linkable(controller): return 0

    # Test fitness at each level
    fitness = 0; similarity = 1; structs = [controller]
    level = 1
    while similarity > 0.6 and level <= 5:
        # Do noisy test
        sim = get_sim(structs, pairs_per_level * 2^level)
        sim.set_input("${50-300},G"); sim.step(4000)
        structs = get_structs(sim)
        similarity = avg_similarity(structs, 2^level)
        fitness += similarity; level += 1
    return fitness

# Quick analysis of whether the port transitions can fire
def is_linkable(controller): ...
# Gets a simulation environment of the correct size with the
# correct structures
def get_sim(structs, size): ...
# Calculates the avg similarity measure for units
def avg_similarity(structs, size): ...
# Returns the structures in the simulation
def get_structs(sim): ...
```

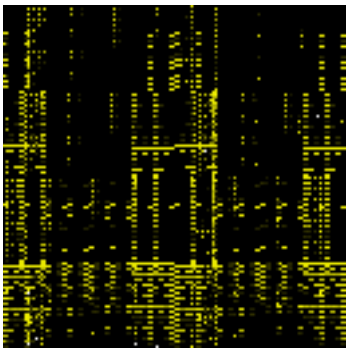


Figure 6.9: The fitness landscape of a 2p/3t assembling unit. Each pixel corresponds to a binary string, counting from 0 to $2^{14}-1$, left-to-right, top-to-bottom. Brighter yellow pixels indicate better fitness (all fitness is normalized between bright yellow and black). White pixels (the brightest) are genomes which when translated evaluate to the maximum fitness found. Overall the image can be simply read as brighter is higher fitness, and the best is white.

of Figure 6.4, tested under the new fitness function, is approximately 2.1. This corresponds to three successive recursive pairing levels, which is fewer than was reported using the original fitness function. It is important to remember that the new fitness value is a weighted average and so fitness tests with more and less recursive levels of pairing are combined to create this value. The stochastic nature of evaluation means that the recursive pairing level varies between fitness tests (depending on how the environment pairs units). Without averaging multiple tests together (as was the case in the original pairing experiments) higher fitness values are observed.

Low dimensional landscapes

Given the modified fitness function described above, the fitness landscape can be directly visualized for low-dimensional assembling units. Such a view provides information on the general structure of the landscapes of recursive assembly. Every possible adjacency matrix and marking of a C/E net can be mapped to a binary string - a process used directly as the genotype encoding for the evolved pairing assemblers - and the fitness values plotted as color in a binary-encoded image. Though the number of genomes grows exponentially with the number of places and transitions, for small values of these the full fitness space is not too large. For 2p/3t controllers, for example, $2\times 3\times 2+2=14$ bits characterizes all the possible controllers, which can be displayed in a small bitmap of size $2^7\times 2^7$. At a minimum, three transitions are necessary for the input and port transitions, the two places are (as always) internal. The resulting bitmap is shown as Figure 6.9.

This initial mapping of the genomes to pixels in Figure 6.9 is arbitrary with respect to how evolution sees the landscape, since numbers close in standard binary counting do not generally have similar bit strings. For example, 7 and 8 in a 4-bit binary code are 0111 and 1000, complete complements of one another. Evolution, however, using low mutation rates, will almost never mutate a genome of 0111 to 1000, and therefore these genomes should be far away from one another when drawn as pixels. A different ordering is needed of the genomes to reflect this idea of evolutionary distance, in order to gain an intuitive idea as to how the fitness landscape is shaped.

A better view can be had using a Gray code (Gray, 1953), which is a well-known method of

0000	0001	0011	0010
0100	0101	0111	0110
1100	1101	1111	1110
1000	1001	1011	1010

Table 6.2: Example of a Gray-coded fitness landscape, where $n = 4$ bits. Each pixel corresponds to a Gray-coded binary string, counting from 0 (0000) to $2^n - 1$ (1000). Counting starts in the upper left corner and winds left-to-right, right-to-left until ending in the bottom left corner (for even rows). Along each axis the bit strings differ by only a single value, and folding the landscape successively in half preserves this property in the new dimensions.

binary counting (though it may be generalized) in which each successive number requires only a single bit change to realize (with wrap-around). Instead of counting to four in binary via 00, 01, 10, 11, for example, there is a corresponding 2-bit Gray code of 00, 01, 11, 10. Because evolutionary algorithms using mutation tend to shift only small numbers of bits at a time, if the binary genotypes of Figure 6.9 are rearranged in the order of a linear 14-bit Gray code, nearby genotypes will also be nearby in mutation space.

One can go further: because of the symmetries inherent in the reflective nature of the Gray code, folding the linear mapping into an appropriate two-dimensional box ensures the genotypes in the second dimension will also be nearby, within one bit change. One can think of the 2D Gray-coded image as a useful projection of how evolution sees the fitness space, which is increasingly approximated by "folding" the image repeatedly in half (touching symmetric pixels together) to imagine higher-dimensional links. Table 6.2 shows an example 4-bit Gray code mapping on a sample fitness space. Nearby bit strings differing by a single bit correspond to C/E adjacency matrices which differ by only a single token or edge. The data from Figure 6.9 is replotted this way as Figure 6.10, as are subsequent fitness landscapes.

By putting similar binary strings closer together, it can be easily seen that the initially scattered high-fitness genomes of Figure 6.9 are actually all part of only a few high-fitness regions. These regions tend to also be symmetric across the center of the space, and would join after folding, indicating that the better fitness is clumped together into regions. Higher-dimensional examples are possible as well, as shown in the 2p/4t example of Figure 6.11 and the 3p/3t example of Figure 6.12.

These landscapes can also be more rigorously analyzed for general ruggedness using the autocorrelation measure, introduced in (Weinberger, 1990) and used widely as a tool for investigating fitness landscapes. Traditional use of autocorrelation is as a measure of how a time-delayed signal correlates with the original, but this idea can also be applied to a mutationally-delayed fitness sequence. By taking random walks through the fitness landscape via some genome-changing operator, it is possible to estimate the correlation in fitness between points at different mutational removes. For smooth landscapes, the autocorrelation can be expected to change slowly with distance, while for rugged landscapes the correlation drops off more quickly. If there is no autocorrelation at any distance > 0, no information is gained about other points by sampling and so no search method will be more effective than random search. For a random walk $\vec{f}$ on a stationary

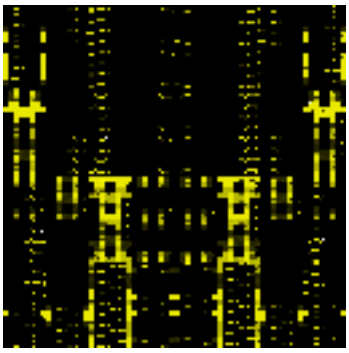


Figure 6.10: The fitness landscape of a 2p/3t assembling unit mapped using a 14-bit Gray code ($2^7 \times 2^7$ pixels). Colors are as in Figure 6.9, and the pixels assigned to each binary genotype are determined as in Table 6.2.

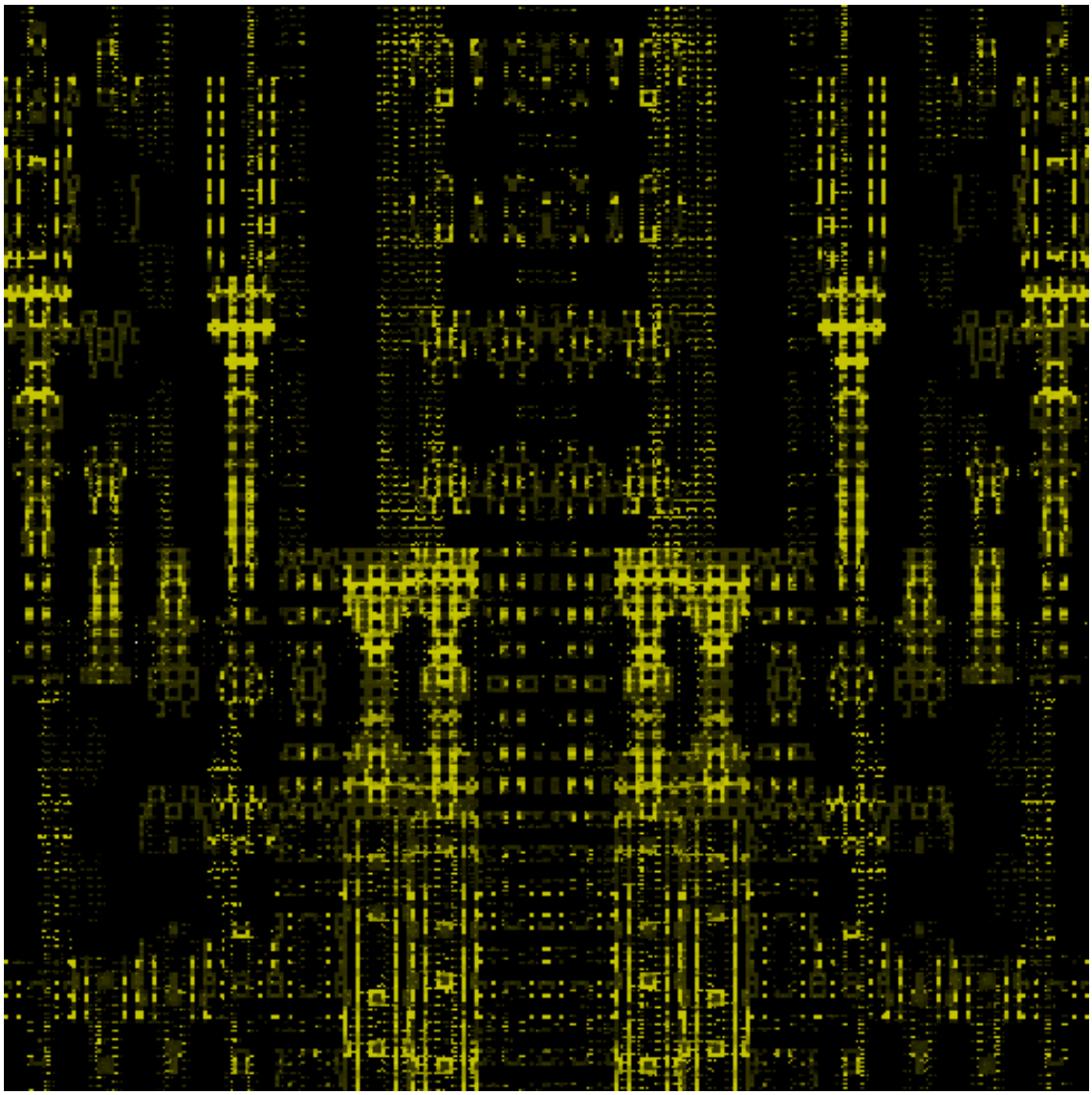


Figure 6.11: The fitness landscape of a 2p/4t assembling unit mapped using an 18-bit Gray code ($2^9 \times 2^9$ pixels). Colors are as in Figure 6.9, and the pixels assigned to each binary genotype are determined as in Table 6.2.

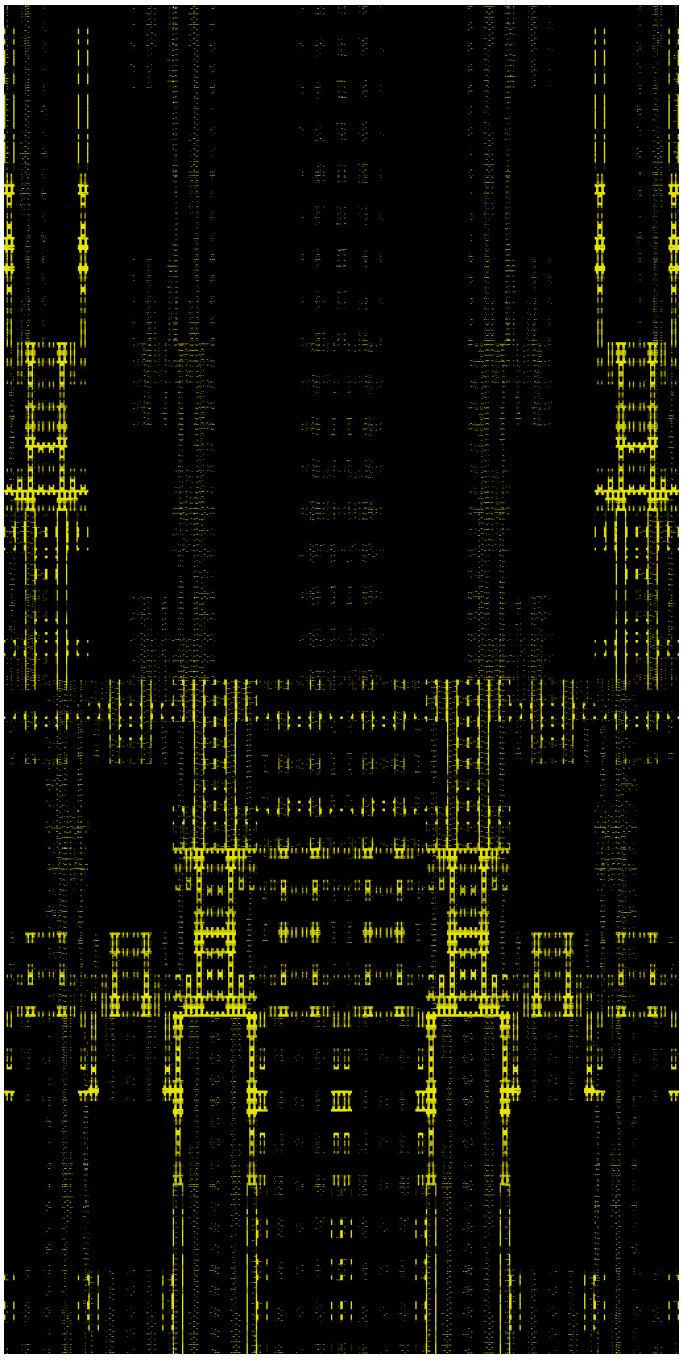


Figure 6.12: The fitness landscape of a 3p/3t assembling unit mapped using a 21-bit Gray code ($2^{10} \times 2^{11}$ pixels). Colors and pixels as in Figure 6.10.

fitness landscape, the autocorrelation r at distance d is:

$$r(\vec{f},d) = \frac{\sum_{i=0\ldots(|f|-d)}(f_i - \mu_{\vec{f}})(f_{i+d} - \mu_{\vec{f}})}{\sigma^2_{\vec{f}}}$$

The values $\mu_{\vec{f}}$ and $\sigma_{\vec{f}}$ are the mean and standard deviation of all fitness values in the walk. Autocorrelation values of fitness landscapes for the recursive pairing task as controller size becomes increasingly large are shown in Figure 6.13. As can be seen in the figure, mutation amounts relative to the genome size result in steeper fitness variation as the genome size grows, indicating the landscape grows more rugged w.r.t. mutation as the binary genome size increases. In addition there seem to be many more non-functional genomes, indicated by the higher baseline autocorrelation value. Unfortunately these genomes are not instrumental in finding good solutions, since the nature of the fitness function ensures they provide no gradient.

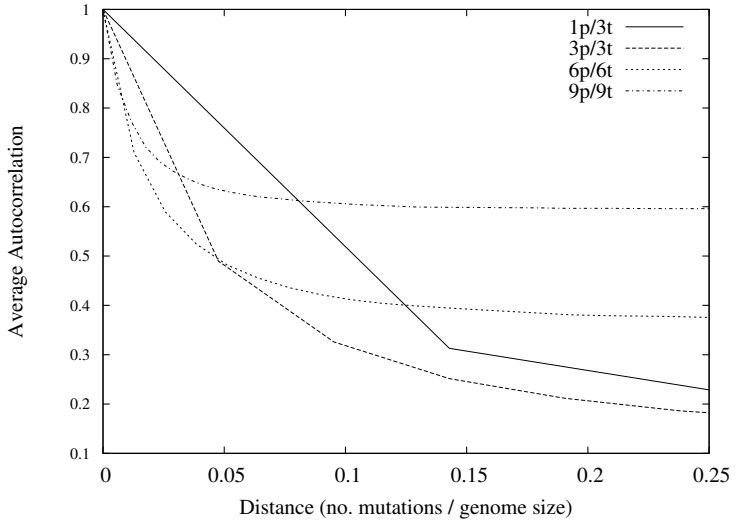
6.5 Tests of other algorithms

A second set of evolutionary experiments was designed to investigate the evolvability problems of Section 6.2, taking advantage of the gradually-scaled fitness function and insights from the fitness landscapes above. As seen from the autocorrelation measure, the relatedness of nearby solutions drops off extremely quickly with more genome mutations, quickly reaching a baseline value. It was hypothesized that mutational algorithms relying less on crossover might do better in this sort of landscape, since larger jumps in genome are statistically random. As was seen in the low-dimensional fitness landscapes, many high-fitness regions also are linked via similar-fitness manifolds, i.e. neutral networks. The 1+1 evolutionary algorithm by (Barnett, 2001) is designed explicitly for these types of landscapes, and so it is reasonable to assume it might also do well for the recursive pairing task. Along with these algorithmic choices, tuning the genotype encoding can also make many landscapes more evolution-friendly.

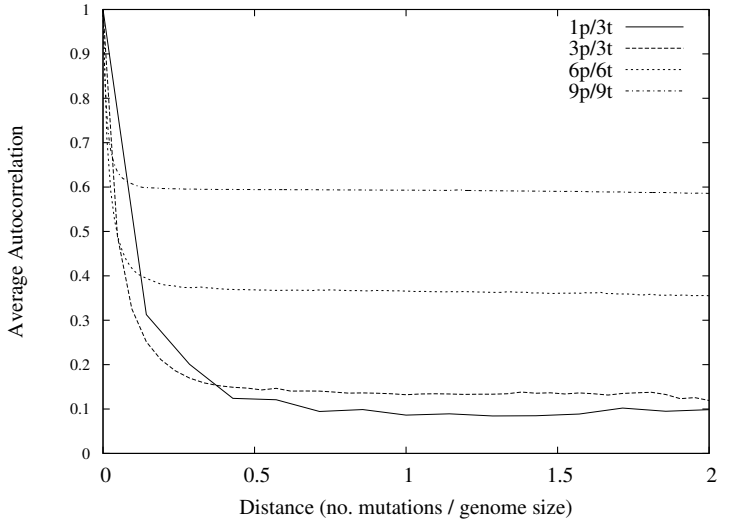
In the next section, the recursive pairing experiments of Section 6.2 are repeated while varying the algorithm and genotype. The goal of these experiments is twofold - to determine algorithms which improve upon the best recursively pairing assembler from the previous study, and as a basis for future work exploring more complex recursive behavior via evolutionary algorithms.

6.5.1 Messy genotype encodings

Previous evolutionary optimization work used a tuple-list-based genome encoding (Kitagawa & Iba, 2003; Nummela & Julstrom, 2005) with some success. Instead of a raw binary string representing the adjacency matrix of a C/E net directly, the matrix is represented in sparse form using a list of (*source*, *target*, *true*|*false*) triples. Each triple represents a C/E net controller edge between a place and transition (or vice versa) being either present or absent. Edges may be over- and under-specified, and later genes (ordering unrelated to source or target) always supersede previous ones. Token markings are represented using triples which have place source nodes and a target= *null*. The idea is similar to that of *messy genetic algorithm* (Goldberg et al., 1989; Mitchell, 1998), in which both a gene locus and value are specified. In the case of two-dimensional Petri nets, the locus is the *source*, *target* and the value is *true* or *false*. Because previous genes may be superseded



(a)



(b)

Figure 6.13: Autocorrelation of the fitness landscapes of recursive pairing assembly using increasingly large controller sizes and binary genomes. The chart (6.13a) is a zoomed-in version of the data from (6.13b). The plotted autocorrelation values are calculated by averaging over the autocorrelation values from 1000 random walks, with walk distance plotted relative to genome size to emphasize how fractional mutation affects genomes of different sizes. As can be clearly seen in (6.13a), as the number of transitions and places grows the relative mutational change results in steeper variation of the fitness values. The higher "baseline" correlation probably results from larger proportions of entirely non-functional genomes, which correlate highly.

but still present in the genome, it was hypothesized that interesting effects may occur where useful substructures can be deactivated temporarily but emerge later.

A different interpretation of the messy genomes is also possible, however. Instead of later genes overriding earlier, conflicting *true/false* edges can be interpreted as edges the genome is "unsure" about. If multiple fitness tests are used, these edges can be initialized to random values each time, resulting in an average fitness which is taken from not a point but a small portion of the fitness landscape. It was hypothesized that this fitness "smearing" might make it easier for assemblers to evolve since adding or removing a single edge often results in very different behavior. The stochastic interpretation of an overspecified messy gene might smooth these edges out, and provide intermediate fitness behavior for otherwise broken assemblers.

Mutation and crossover for messy genomes (probabilistic or ordinary) is accomplished here in a way analogous to two-point binary crossover for a MGA. A random two-point chunk of genes from the successful genome is used to overwrite genes in the less successful genome. After crossover, mutation changes a number of genes n by removing r genes and adding a genes such that $a + r = n$, where n is chosen by flipping a biased coin for each gene.

6.5.2 Other evolutionary algorithms

Random mutation hill climber

In addition to the change of genome encoding, the use of other evolutionary algorithms was also hypothesized to lead to better overall results. The MGA operates in a way similar to traditional genetic algorithms, using the standard notions of crossover and mutation (though in a particularly simple form). By the no free lunch theorem (Wolpert & Macready, 1997), these operators are not ideal for exploring all types of fitness landscapes. Results in genetic programming (Banzhaf et al., 1998; Koza et al., 2005) have suggested that particular types of crossover are not useful, or actively disruptive, and Petri net evolution can be viewed as an instance of genetic programming. The standard MGA has a tunable recombination rate *REC* or ρ (Harvey, 2009), which is the percentage of genes which are "infected" to the losing genome after each tournament (set to 0.5 in our previous work). By setting this recombination rate to zero, the MGA becomes in essence a parallel random mutation hill-climber (RMHC).

1+1 netcrawler

Work by (Barnett, 2001) also shows that when significant neutral networks are present in a fitness space, non-parallel exploration via a 1+1 netcrawler is optimal. The landscapes above show significant neutral areas, so this type of search algorithm was also hypothesized to be an improvement on the standard MGA.

The coevolutionary MGA

A third approach to evolving Petri net controllers is one of the earliest explored - coevolving a population of *transitions*, not Petri nets, which are then combined with one another to form complete Petri net solutions (Reid, 1998). In this work, fitness is assigned to each unit based on the fitness of the complete Petri nets constructed from that unit and others, similar to the original bucket brigade of (Holland, 1992). Here we present an elegant extension of the MGA, named the CoevMGA, which allows the same type of coevolution to occur.

Algorithm 5 The coevolutionary MGA in Python pseudocode.

```
def evolve_coevmga(pop_size, num_evaluations, genotype_size):

    # Create the initial population
    population = [init_genes() for i in range(0, pop_size)]
    fitnesses = [[] for gene in population]

    for i in range(0, num_evaluations):
        # Create a genotype out of many sets of genes
        genotype = [rand() * len(population)
                    for i in range(0, genotype_size)]

        phenotype = translate([population[gene] for gene in genotype])
        fitness = fitness_function(phenotype)

        # Store the fitness back to each of the genes
        for gene in genotype:
           fitnesses[gene] = fitnesses[gene] + [fitness]

        # Partially overwrite one gene, as in the MGA
        winner, loser = select_genes(population, fitnesses)
        population[loser] =
           partial_overwrite(population[winner], population[loser])

def init_genes():...
def fitness_function():...
def select_genes(population, fitness):...
def partial_overwrite(win_genes, lose_genes):...
```

The CoevMGA is a variant of the standard MGA (Harvey, 1996) which was designed to allow the coevolution of partial phenotypes to solve optimization problems, particularly the evolution of C/E nets. Instead of a single genome translated into a single phenotype, many sets of genes are combined together in the MGA to make a genotype, which is then translated into one phenotype. The fitness of the tested phenotype is then distributed back to the many component genes of which it is composed. In all operations except fitness evaluation, the CoevMGA behaves identically to the standard MGA.

The full algorithm is listed as Algorithm 5, with additional functions dependent on the particular implementation used. In addition to the normal MGA population, an additional list of fitnesses is maintained for each set of genes in the population. For each evaluation, a genotype is constructed from genes, translated into a phenotype, and is then passed to a domain-dependent fitness function. The resulting fitness is then added to each of the component gene fitness lists. Depending on the gene selection function, it may be possible to replace the fitness list with an average value or maximum value.

Experimental Setup

In summary, the genotype encodings and algorithms hypothesized to enhance the evolution of scalable assembly are listed below along with the original binary genomes and MGA. Parameters of the encodings and algorithms are set to make comparisons as straightforward as possible. The

evolved controller size is held at the optimal size of the previous evolutionary runs: 7p/10t (representable in $2 \times 7 \times 10 + 7 = 147$ bits). By using the previous evolutionary runs as a benchmark, the effect of the new scaling fitness function can be directly observed. This then provides a baseline from which to compare other algorithms and encodings.

Encodings:

- Binary encoding
  - Initial genomes of 147 bits with approximately 10% bits initially set (this was found to provide significantly faster convergence on interesting solutions)
- Messy encoding
  - Initial genomes with 15 (*source*, *target*, *true|false*) triples (10% of $147 \approx 15$)
- Stochastic messy encoding (ProbMessy)
  - Initial genomes with 15 (*source*, *target*, *true|false*) triples

Algorithms:

- Microbial GA (MGA)
  - Population size 200
  - Two-point crossover, $\rho = 0.5$ recombination rate
  - 5% of genes per genome mutated on crossover
  - 10 independent fitness tests performed per genome evaluation
  - Fitness is the weighted mean fitness value over all fitness tests, weighted as described in Section 6.4.1.
- (Microbial) random mutation hill climber (MRMHC)
  - Population size 200
  - No crossover, mutation as above
  - Fitness tests as above
- Coevolutionary MGA (CoevMGA)
  - Population size 200
  - Crossover and mutation as above
  - 100 random genomes (half the population) combined to create a full C/E controller for each evaluation, genomes initialized with 1% of bits initially set or as a single messy gene for the messy gene encodings; binary genomes combined by OR'ing them together, messy genomes combined by addition of gene sets
  - Fitness tests as above, the resulting fitness value is recorded for each genome that was combined to form the full controller
  - Genome fitness is the average fitness value over all C/E controller evaluations of which it has been a part
- 1+1 netcrawler

- 1 netcrawler in population
- Mutation rate auto-adjusted every 10 evaluations so that neutral mutations occur with probability $1/e$
- Neutral moves defined as exploring a genotype with fitness ± 0.2 from the known genotype
- Fitness tests as in the standard MGA

The simulation environment was tested with transition firing rate $\tau = 5$, and assembly rate $\alpha = 1$ assembly event per time step.

6.5.3 Results

Evolutionary results comparing the above encodings and algorithms are presented as Figures 6.14 to 6.16. 15 evolutionary runs were performed for each set of parameters, and results combined as described in each figure. The source code for the newer evolutionary tests is also available at:

- https://coralassembly.wordpress.com/

As was discussed for the previous recursive pairing tests, when better pairing solutions are found the evaluation speed drops significantly, since fitness evaluations become much more expensive. Consequently, comparing algorithms and genomes in terms of number of evaluations is misleading, since much more or less calculation may have taken place per-evaluation to reach a particular fitness value. For example, the evolution of binary genomes tends to generate proportionally more zero-fitness individuals, but these individuals take almost no time to test, and so binary-encoded generations seem to progress much faster using the raw evaluation metric.

Ideally the algorithms could be compared directly using computation time, but across shared computers this is difficult to measure directly. Since the evaluation time is dominated almost completely by the execution of the simulation environment, a useful proxy is the number of events executed by the simulation. These events are of semi-constant time, each corresponding to a simple C/E net, input, or assembly action, and are the primary computational load of evolution. In measurements taken while evolution was occurring, the time to execute a thousand events (or kEvent) remained largely constant per-run and between runs (approximately 0.001s or 1ms on the computing cluster used). The results below plot fitness vs. kEvents, which should be read as a measure of how fast the fitness of evolved pairing controllers increases using various algorithms and genotype encodings. Each evolutionary run was recorded over 10^8 kEvents $\approx$ 28 hours.

Best assembler

The best assembler, found by the 1+1 algorithm using binary genomes, is shown as Figure 6.17. The most notable difference between this unit evolved during the benchmarking runs and the previously evolved assembler of Figure 6.4 is the different connectivity to port and signal transitions; these edges are now evolved and may connect to multiple places. This ability was exploited to achieve higher fitness, though the overall behavior of the assembling unit is qualitatively very similar to that of the previous assembler.

The pairing behavior of the best benchmark assembler also has three core states: *initial*, *decide*, and *pass-through*. The initial state, shown in Figure 6.17, enables both *In*- and *Out*-ports

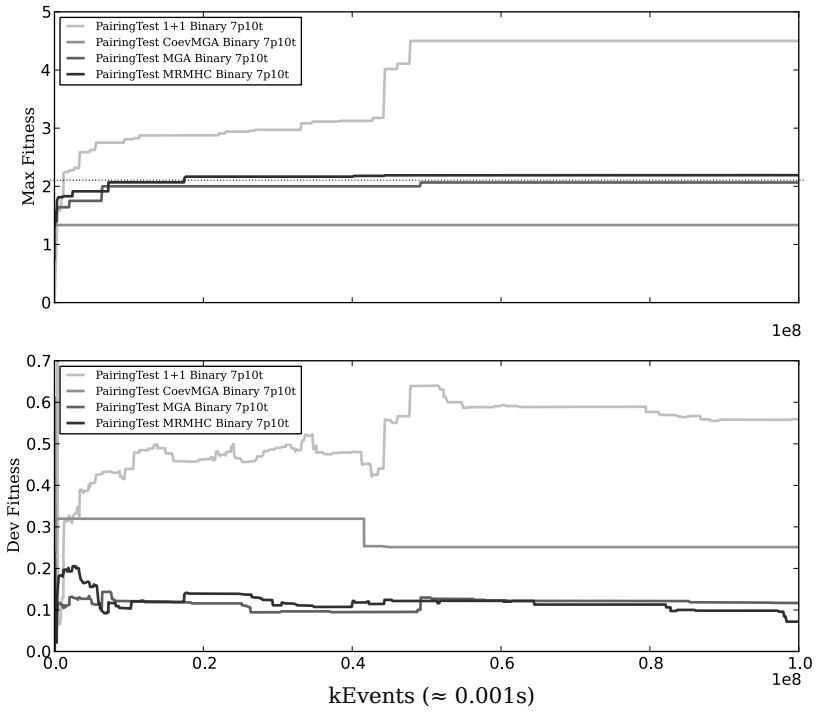


Figure 6.14: Results evolving binary genotypes with various types of evolutionary algorithms using the new recursive pairing fitness function. The upper chart indicates the maximum fitness found by the evolutionary algorithm over all of the runs, while the bottom shows standard deviation in the maximum between runs. All results were recorded over 10^8 kEvents $\approx$ 28 hours. The thin dotted line represents the fitness of the best pairing assembler found in the previous tests (Figure 6.4).

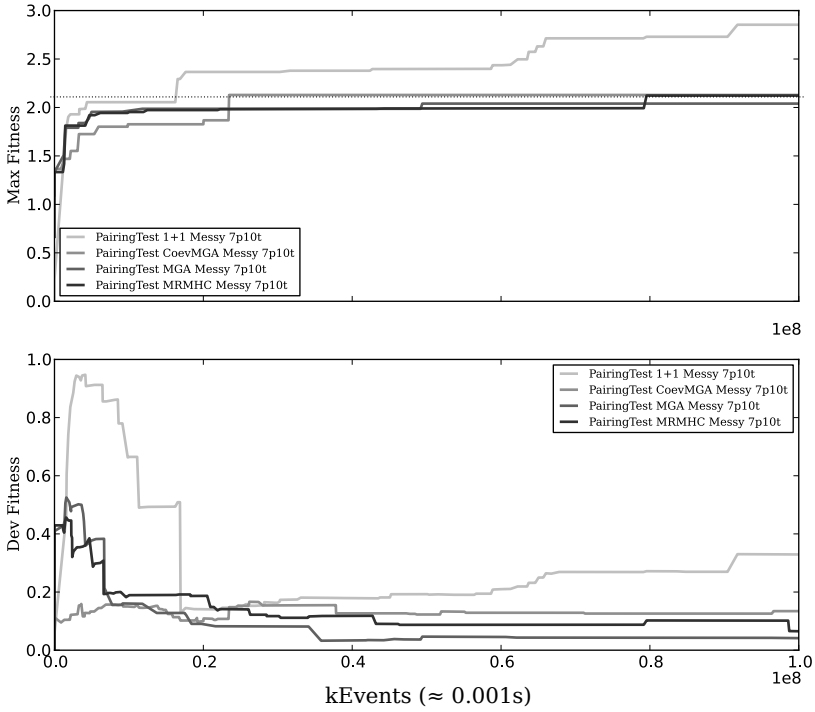


Figure 6.15: Results evolving messy genotypes with various types of evolutionary algorithms using the new recursive pairing fitness function.

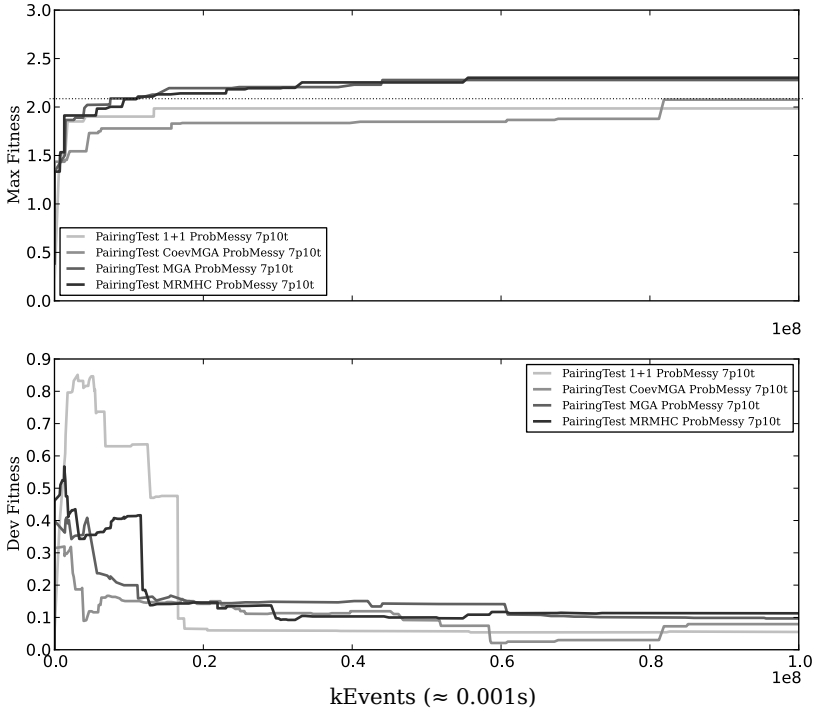
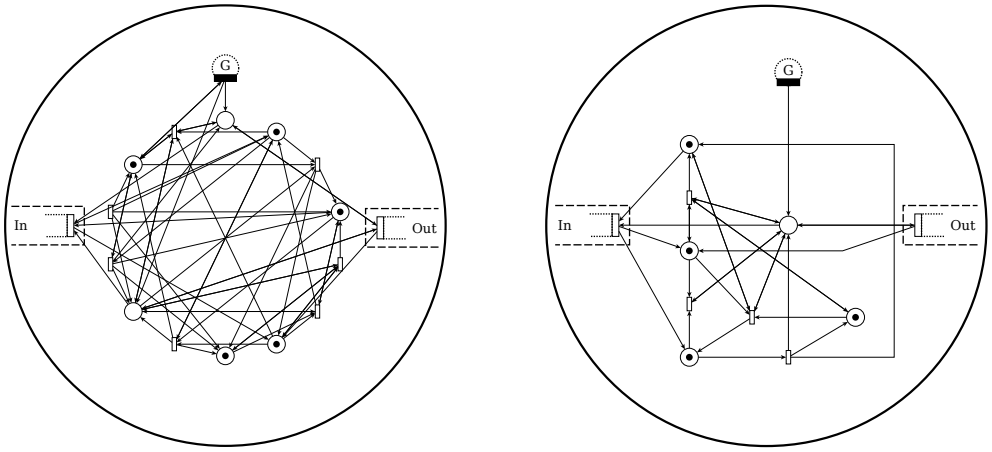


Figure 6.16: Results evolving stochastic messy genotypes with various types of evolutionary algorithms using the new recursive pairing fitness function.



(a) (b)

Figure 6.17: Best final evolved assembler from the benchmark evolutionary runs (6.17a), found using the 1+1 algorithm using binary genomes (as shown in Figure 6.14). Extra places and transitions which do not fire or are duplicates were removed and merged to create the simplified version of (6.17b). Unlike the evolved pairing unit of Figure 6.4 above, edges to the signal and port transitions are also evolved, allowing for various types of synchronized transitions between units when linked.

in response to a *G* signal, so that a given unit can connect on either the left or right side. This does not lead to long chains and rings of units paired at both ports, since individual units are capable of atomically disabling one port when another is connected. In addition, since each unit has both ports open, there is no risk that a stochastic choice of *In* and *Out* port will partition the units unevenly. As a consequence, full pairing always occurs with individual units, though units may pair with themselves, reducing fitness. Extended multi-unit structures cannot share state, however, which makes disabling the opposite port when connected impossible as an atomic action, so this technique is only useful for the initial pairing operation.

The behavior after the first pairing step is essentially identical to that of the previous evolved assembler. Once linked, the *In*-port connected unit of a pair structure has a *decide* marking, while the *Out*-port connected unit operates in *pass-through* mode. In general, even after multiple pairing steps in response to further *G* signals, the rightmost unit in a chain of units will always *decide*, while all other units (in *pass-through* state) pass tokens back to the leftmost open *In*-port. The *pass-through* behavior is faster, since in these newer benchmarking evolutionary runs it is possible for both port transitions to share common input and output places. Thus the firing of the *Out*-port transition can directly enable the *In*-port transition without an intermediate state change (which otherwise requires an additional transition firing). The rightmost *In*-port at the end of a *pass-through* chain is enabled approximately twice as fast due to this optimization, resulting in better pairing as the chains grow and higher fitness.

6.6 Remarks

Of the evolutionary algorithms tested, the above results show clearly that the 1+1 netcrawler algorithm using binary genomes finds better recursive pairing solutions, faster, than all the other tested

algorithms. Using a messy encoding does not appear to help find higher fitness more quickly, and the ProbMessy encoding helps even less, though it is noted that in select runs allowed to exceed 10^8 kEvents the 1+1 algorithm with standard messy encoding approaches the maximum fitness of the binary encoding. Aside from the 1+1 algorithms using messy and binary genomes, all benchmarked combinations found solutions similar in fitness to the previously evolved best assembler.

The MGA with and without crossover (MRMHC) perform almost identically in all three runs, which indicates that the MGA's infective crossover is of little to no use in finding good recursive pairing solutions. In general, the population-based methods do not seem to benefit from parallelism when compared to the 1+1 netcrawler, and the overhead generated by evaluating many semi-fit solutions seems to overwhelm any gains of sampling in different areas of the fitness space. This is somewhat reflected in the higher fitness deviation for the 1+1 netcrawler when using binary and messy genomes. It seems the population-based methods are more consistent, particularly when using binary genomes. Given a fitness function with constant evaluation time, a population-based approach may be more effective. Using approximation methods instead of direct simulation may provide this kind of speedup, and this idea is discussed as future work in Chapter 8.

It was hypothesized initially that the coevolutionary MGA might evolve better solutions by combining multiple useful sub-nets, but this was not found to be the case. The CoevMGA performed more poorly than other methods for all genome encodings, and performed particularly badly given binary genomes. The combination method for binary genomes (OR) may not have been appropriate for good solutions to be found, since sparser genomes tended to evolve more effectively and OR'ing multiple bit strings together only increases the number of set bits. In addition, fitness is assigned equally to all contributing genotypes with no bucket-brigade as in (Reid, 1998), which may be a necessary factor for better coevolution.

The recursively assembling unit which paired more effectively than the previous best evolved result was found by the 1+1 binary benchmarking test. However, despite the speed increase enabled by more flexible evolved connectivity to port transitions, the new evolved unit uses effectively the same qualitative methods to control recursive pairing as the previously evolved assembler. No alternation of open ports is observed, for example, and port activation seems to flow in only a single direction through the chain structures. This may reflect limitations on units' ability to process information given the amount of C/E net components available, and future tests using larger genomes and the more effective 1+1 algorithm may find more interesting behavior. Alternately, the use of more complex environmental signals or ports may make the task easier, and coevolving the signals and/or unit architecture may be a useful methodology applicable to more complex scalable assembly tasks.

In conclusion, the automated design of scalable assembling systems is a challenging project, and requires tools from a variety of related areas. In the next and final chapter various directions for future evolutionary study are discussed in context of the designed assemblers demonstrated previously. Lessons learned during the construction of these scalable and controllable models are applicable to future evolutionary work, and vice versa. As must be the case, the benchmarks above investigate only a small portion of the diverse methodologies for evolutionary design, and point generally toward other methods which may also yield good results.

Chapter 7

Principles of Recursive Assembly

> A well-known scientist (some say it was Bertrand Russell) once gave a public lecture on astronomy. He described how the earth orbits around the sun and how the sun, in turn, orbits around the center of a vast collection of stars called our galaxy. At the end of the lecture, a little old lady at the back of the room got up and said: "What you have told us is rubbish. The world is really a flat plate supported on the back of a giant tortoise." The scientist gave a superior smile before replying, "What is the tortoise standing on?" "You're very clever, young man, very clever", said the old lady. "But it's turtles all the way down!" (Hawking, 1988)

In the above quotation from Stephen Hawking's *A Brief History of Time*, science is humbled by an appeal to a recursively composed system. Though this particular example is meant to be amusing, it is a natural idea that systems are built on top of others; it is the way in which most if not all scientific knowledge is organized (perhaps an irony not lost to Hawking). Biological and chemical systems abound with structures easily categorized in this way, often built through processes of assembly. Harnessing these types of systems and creating our own artificial examples has proved challenging, however, even with modern advances in computer and chemical hardware. While it is generally agreed that incredible potential exists even in our current abilities to create assembling systems at nano, micro, meso, and macro scales, a general process or framework is missing in which these systems can be designed and controlled while *bridging* these scales.

The problem is a classic question of artificial life, where interesting abilities inspired by natural systems are restated as computer science to be better understood and engineered. Often this takes the form of a prototypical model, providing a constructive basis for generating other, artificial forms of the behavior. The work in this thesis is largely undertaken in this spirit, with the goal of generating simple examples of realistic but computational assembly across scales using recursive behavior. Such abilities will always depend somewhat on the particulars of the assembling units and environment (though care has been taken to ensure that the model is extensible in these areas), but the ability to build arbitrarily large devices in simulation with simple control and realistic non-reprogrammable parts is an important step toward building simple and powerful assembling devices in the real world. The abstract model is also shown to be quite capable - with smaller memory than a single register in a modern CPU it is possible, through distributed assembly, to build any computing device. These prototypes and assembly framework provide a pragmatic,

well-defined basis for future work in the assembly of large and controllable artifacts. Usefully composable atomic primitives are the core part of the assembly problem when formulated this way, and automated search for such primitives has also been explored in this thesis.

7.1 Recursive assembly

As mentioned in the introduction, the goal of the CORAL model is to provide a plausible mechanism for unlimited hierarchical assembling behavior in a realizable but computationally strict system. Traditional state-of-the-art assembling systems and models have to date largely focused on organization at one or two nested levels, often requiring qualitatively different mechanisms for structure formation and interaction as assemblies grow larger. This is often the case in natural systems, but it complicates and makes incomparable the basic definitions of structure and interaction across competing models. As discussed in Chapter 2, more formal computational models of assembly also exist, such as cellular automata or artificial chemistries of computational units. These formalisms, while generally more successful at generating multi-scale interactions, have tended to ignore important constraints on realistic systems, such as conservation of mass. This has strongly limited the application of this work to real systems, despite the interesting results (particularly in self-replication).

The importance of control over dynamic hierarchies has also only recently become more appreciated as roboticists and chemists gain the ability to engineer powerful assembling units. There is broad recognition that the standard, centralized control paradigms do not scale to large numbers of units. A common metaphor for the problem is that of the cell and the body, which is a profound example of organization at many scales with highly flexible behavior. Biological cells are self-contained units containing all the information needed to regenerate a full system, and it is obvious by our very existence that such units have great potential. There is less recognition, however, that the cellular metaphor obscures important aspects of our organization related to a developmental view of an organism which is critical for the control and design of related artificial devices. There are no DNA sequences encoding the exact cellular positions in the tissues of our arms, for example, these are created implicitly via a mediated process of growth. When we wish to move our arm, it is not required that our brain or any other single organ compute the movement of each muscle fiber (or perhaps even muscle groups), the relatively simple neural signals are converted into motion via the *structure of the arm itself*. The fact that each cell still contains (more or less) the full genetic code is largely irrelevant to the fact that the computational structure of our arm and the rest of our body was built through assembly via chemical signals years ago, not necessarily arising even from our own DNA.

The idea becomes much more starkly defined when individual units themselves are entirely incapable of storing the structure of the whole. The same type of directed assemblies can be built, but now there is an explicit need for some sort of external control. Proteins are a better example of this type of assembly - a cell itself is also a highly complex multi-scale machine, but as opposed to the body only one “part” stores a symbolic copy of the whole, all other information is stored implicitly in the current chemical and physical organization of the system. The protein paradigm has significant advantages when applied to engineered systems: the atomic parts are *much* simpler, while control remains highly distributed. The key aspect of artificial assembly is then transformed

into finding appropriate primitives as near as possible to the threshold of complexity at which individual units can be assembled into useful structures at many scales.

The CORAL model was designed explicitly to explore this boundary in a computational setting. As was shown in the final section of Chapter 3 and results evolving pairing assemblers from Chapter 6, most choices of computational primitive have only limited ability to generate controlled structure over multiple scales. Well-mixedness and the inability of units to directly affect environmental interactions (realistic for many systems) make the issue worse, since as discussed in Chapter 5 non-tree assembly requires complex processing for even near-deterministic results. Of those units where assembly does not become totally random, the structures formed are generally limited in scale or non-responsive to external control. The NOR units, C/E units, and evolved assembling units of Chapters 4, 5, and 6 avoid these pitfalls, while demonstrating recursively scalable and computationally powerful assembly.

7.2 Why recursive assembly?

While this thesis and the above discussion switch between descriptions of scalable and recursive assembly rather freely, it is not necessarily the case that a scalable assembly process need be recursive. A well-defined incremental growth process may be capable of building arbitrarily large structures, and there are many examples of simulated and real systems with these properties (particularly robotic systems), including the C/E unit. There is no need for structures of particular "granularity" when talking of incremental assembly, extremely complex structures may be specified piecewise or otherwise. Recursive assembly does have a number of interesting tradeoffs, however, which make it the primary target of this work. It is difficult to explain fully the ideas without a concrete example in mind, however, which is why this explanation has been diverted until after the introduction of several assembling units.

The primary motivation behind recursive assembly is derived from the assumption of an environment with only an external broadcast signal (the assumption of the CORAL model). This signal has unlimited reach but highly limited bandwidth - only one signal at a time, no way to address individual units. Under these assumptions, imagine a new device is required from a system of units. One traditional method of assembly for programmable devices is to send the full device specification via the signal to the devices, which then assemble themselves using some distributed algorithm. Units must have high memory and lookup ability to store large specifications, however, so this method does not scale downward easily. A second method is to develop an algorithm and unit which allows the structure to be built piecewise (one piece at a time, since signal bandwidth is limited). The size of the pieces may vary given the input and units, but with very large structures this results in longer assembly times.

7.2.1 Brains and bodies

A key observation emphasized by this thesis is that, at a given scale, structures made of many parts usually have a small number of complicated portions and very many other, primarily (but not entirely) structural, components. This can be seen in the design of cells, organisms, robots, and city architecture, to name only a few examples. A walking robot, for example, might have a complex microchip brain at the microscale (or smaller), but legs of a basically uniform material except for

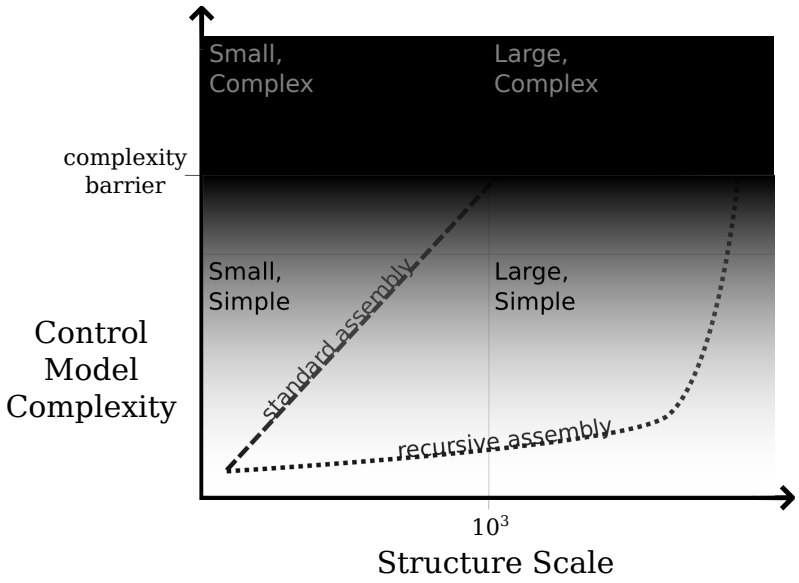


Figure 7.1: Recursive assembly vs. standard assembly, showing the "shortcut" through large, simple structures. Recursive assembly processes decouple complexity from scale, allowing one to "build out" in scale before "building up" in complexity and access new regions of the assembly space.

joints at the macroscale. If the robot is specified in macroscale terms, there is no way to build the brain via assembly. If the robot is specified at the microscale, the specifications will be incredibly redundant and piece-by-piece assembly will take a relative eternity. Recursive assembly provides a way out of this dilemma, by allowing one (in the case of NOR units) or potentially many types of well-defined and controllable meta-units to be built from individual parts. This requires that the parts themselves be structured so that particular forms result in higher-level control; in other words a recursive construct. Importantly, the property above can be an *addition* to an already present one-by-one assembly process, capable of building all forms incrementally. Linking all these ideas together, if the recursive construct has identical capabilities to the primitive assembling units which are able to build any structure incrementally, arbitrarily large structures can be built much more efficiently by building "out" (recursively) as far as needed for the scale required and then building "up" the required computation and structure (see Figure 7.1). Assuming the structure is patterned appropriately, this can result in huge information *and* control compression. Resuming the previous robotic example, even if specified microscopically a robot will be useless unless commands can be issued for *macroscopic* behavior. Recursive constructs can again be exploited for this purpose, allowing parts of the robot built from microscopic parts to be modeled at high granularity, despite the fantastic complexity, since the *structure itself* ensures behavior will be simple and predictable.

NOR units are a concrete demonstration of this idea. NOR meta-units, of all levels, are also by definition tree structures of individual NOR units. Since the waterfall assembly process of Section 4.3.3 can build structures of arbitrary depth, any meta-unit structure can therefore be built equivalently using no recursive assembly at all. The number of signals sent to the environment is exponentially greater using the incremental method, however, and the signals sent poorly reflect the inner logical structure of the final construction, since many parts of the meta-unit structure are present simply to "cancel out" other portions given particular inputs. Of course, there are

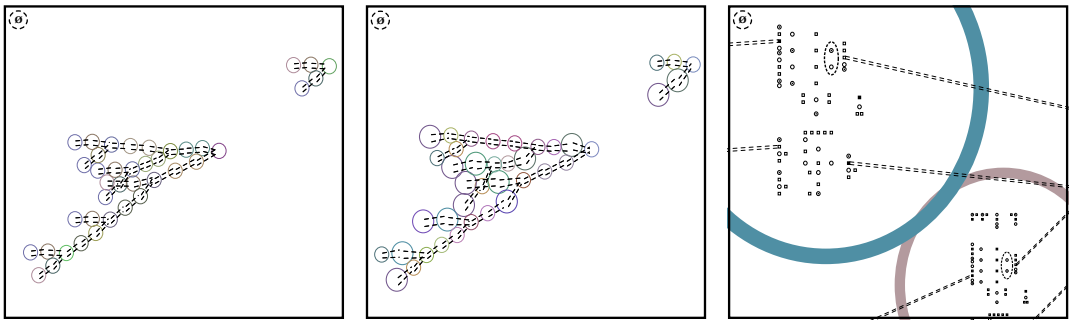
many pathological forms not easily decomposable given this or any other method of recursive construction; for example a large, randomly attached tree of NOR units. These types of structures are probably not compressible by any general means, however, and their control is fundamentally limited by complexity in the same way, making them of limited use.

In summary, the claim is that there will generally be *some* types of exploitable multi-scale patterns inherent to a useful device of many parts, because otherwise such a device would be impossible to understand and control. Even so, given a device of this type, one can choose at any scale to assemble via incremental assembly, so nothing is lost. The core argument is that the computational parts of a large device may be quite complex, but the entire device will likely not be complex at the same scale and this can be exploited via recursive assembly. In an even more cursory statement: brains and bodies build the same, but at different sizes. This fits well with post-GOFAI ideas in AI and robotics where the body is itself a necessary and inseparable computational device of the brain to which it is attached (Varela et al., 1992; Brooks & Stein, 1994; Pfeifer & Iida, 2004; Clark, 2008; Bongard, 2009).

7.2.2 Logical "bodies"

An important consideration which is omitted from the CORAL simulation but highly important in practice is what the computations of structures are "good for." To be interesting as an engineered device, NOR units must be able to perform useful functions aside from recursive assembly, preferably ones that become more useful as structures grow in scale. Without this, recursive units simply attach to one another indefinitely and compute logical functions, and while this is a philosophically interesting example to those interested in organizational hierarchy it may be of less practical use.

As mentioned previously in Chapter 4, NOR units are based on the NOR operation in Boolean logic, from which any sort of Boolean logical expression can be generated. Particular NOR unit structures therefore can perform any Boolean logical operation, the result of which can control any other hypothetical unit action like motion or sensing. It is easy to imagine motion linked to the local output of units, for example the current output state determining whether or not a part expands or contracts like a unit-compressible module (Rus & Vona, 1999; Ishiguro et al., 2006). Figure 7.2 below gives such an example. Though investigating particular types of physical environment is outside the scope of the work in this thesis, many different types of actions could theoretically be linked to the computation of units and meta-units. Because of the recursive structure of meta-units, these actions might also naturally scale, depending on the nature of the environment and the action.



(a) (b) (c)

Figure 7.2: An example of unit-compressible behavior added to a NOR-unit. A NOR unit structure and meta-unit structure are seen in (7.2a). By driving the unit expansion/compression using the output state of each unit (7.2c), controlled patterns of expansion expressible in terms of propositional logic can be generated in the structures (7.2b). This can be exploited for motion or other actions. Due to the recursive functional nature of the NOR unit, expansion behavior for meta-structures is also well-defined and easy to manipulate.

7.2.3 Assembly potential and limitations of C/E units

C/E units are a useful counterexample for the above discussion, since C/E unit structures are computationally more powerful than NOR unit structures and are built on a recursively composable model (the C/E primitive). However, the turtle assembly process itself is not recursive. Arbitrarily complex devices can be built using C/E units, but these devices can only be built incrementally with no recursive speedup currently possible. As will be discussed in later sections of this chapter, this is not believed to be a fundamental limitation of the more powerful assembler. Recursive C/E assembly is probably possible, however it is more difficult to exploit the naturally recursive but more complex computation in substructures for use in assembly. The assembly of arbitrary computing devices in a realistic environment without programmable atomic units is interesting for other reasons, however, and is discussed below.

7.2.4 CE meta-constructions

To date, only smaller meta-constructions have been generated in simulation, though larger structures are possible. The major reason for this is the large number of C/E units and background signals required, which is exacerbated by the need for many time-consuming assembly filtering steps for a structure built with many cyclic connections. As discussed above, creating connections between two parts of a single structure type, a cycle, inevitably results in the creation of incorrectly linked chains and cycles of incorrectly assembled structure instances. Filtering can reduce the number of these incorrect assemblies, but the process is inherently stochastic and requires many repetitions to be confident no incorrect structures remain. For large structures with many edges not captured in a spanning tree the time and signals required can be enormous.

The problem is partially solved if the well-mixed assumption of the CORAL model can be relaxed somewhat. For example, real protein chains, despite folding taking place in a stochastic environment, generally do not have this problem (to the same degree) because identical folding

protein chains separated by a distance will not tangle with one another instead of themselves. This is not necessarily true in all cases, especially as physically flexible structures grow orders of magnitude larger than their component parts. Experience with the C/E unit assembler suggests that locality may be best modeled as a heuristic optimization to the simulation, reducing the number of filtering steps needed but not entirely eliminating the fundamental need for structure verification. Other rule-based assembly models in robotics and combinatorial chemistry take similar approaches, since physical environments are difficult to model both rigorously and efficiently (Yang et al., 2008; Thorsley & Klavins, 2008; Mermoud et al., 2009).

The second major reason for smaller structures, introduced in the previous section, is that while C/E units are able to emulate their own C/E unit *controller* in composition, C/E meta-units do *not* respond identically to primitive C/E units to the assembly commands themselves. In other words, while a C/E meta-unit would behave like a C/E primitive if built into meta-meta-structures (assuming appropriate port connections), C/E meta-units have not been shown to be capable of this assembly themselves. Unlike the NOR meta-units, which can be assembled efficiently into higher-level structures via shielding and a delayed waterfall selection algorithm, C/E meta-units do not "interfere" with themselves so as to respond to signals only from appropriate ports and subunits. Because of this, the exponential control compression for assembly through the construction of meta-units is not currently available. Again, this is not believed to be a theoretical limitation, as the turtle assembly and state-shifting used in standard C/E unit assembly has arbitrary ability to manipulate the state of assembled structures. Unfortunately, the higher computational complexity of assembled C/E units makes it difficult to generalize about the behavior of structures, and any solution will probably be more complex than those designed for NOR unit primitives. Until this is accomplished, meta-unit assembly remains highly suggestive but only of use in controlling, not constructing, C/E structures.

Aside from the recursive limitations, the meta-Petri nets constructable from C/E units are a powerful modeling and control basis for the assembled devices themselves. These devices, being built of many parts, rely on the appropriate synchronization of many distributed interactions and Petri net models were originally designed for these tasks (Peterson, 1981). Feed-forward Boolean logic, as realized by NOR units, provides a very simple mapping of inputs to output with limited dynamic potential, making more complex or multi-step actions difficult to realize in assembled devices. Meta-nets built using C/E units, by contrast, allow one to capture interactive, parallel, and synchronized behavior: anything which can be built as a safe Petri net of any size. Because of the additional computational power, assembled C/E units can form the basis of other Turing complete control abstractions. These custom control models may then be scaled upward and downward as effectively as the meta-nets on which the models are based.

7.3 Models of assembling systems

A major enabling factor allowing the unambiguous demonstration of multi-level hierarchical construction and assembly is the use of the CORAL model. Models of assembling systems are of limited use in investigating these topics when external assembly and/or communication uses a model incompatible with the internal computation of the units themselves. External interactions become internal interactions when studying assembling systems at multiple scales, leading to complex

hybrid models when environmental interactions are not strictly comparable with the internal processing of base units. Artificial chemistry has many examples (Ono & Ikegami, 2001; Ewaschuk & Turney, 2006; Hutton, 2007; Grushin & Reggia, 2008), as does artificial life (Bedau et al., 1997; Ray, 1997) and modular robotics (Rus & Vona, 1999; Kotay & Rus, 2000; Detweiler et al., 2007), among others discussed in Chapter 2. In the CORAL model, the primitive synchronization operations of Petri nets provide the basis for all of computation, assembly, and communication, making assembled structures simply large instances of what could be an atomic C/E net.

In other pure-computation models with similar properties, such as Alchemy (Fontana & Buss, 1996), ε-machines (Crutchfield & Görnerup, 2006), FSM soups (Salzberg, 2007), or the general case of process calculi, there is a lack of physical realism without appropriate constraints because interactions effect large transformations of the units themselves. Physical units often have strict limits as to how interactions may or may not occur, and generally mass cannot be destroyed. The CORAL model makes these assumptions, forcing atomic units to specify *in entirety* beforehand the finite ways in which they can be influenced (through synchronization) by other units. Unlike other realistic artificial chemistries which investigate units designed or reprogrammed for each task (such as the κ model (Danos et al., 2007) or the graph grammar assembly of (Klavins et al., 2006b)), complex reconfiguration in the CORAL model is ideally achieved by solely manipulating information sent to the system. Reconfiguring information is often much easier in practice than replacing or reprogramming highly-engineered hardware, wetware, or molecular units when faced with a new assembly target, especially when such units are limited in memory and/or complexity.

7.3.1 Self-organization and necessary complexity

There has been debate in self-organization literature as to whether continual addition to the underlying interactions is a necessary component of hierarchically organized systems (Rasmussen et al., 2001b; Groß & McMullin, 2001; Rasmussen et al., 2001a). The simply-stated position or *ansatz* of the addition-is-necessary proponents is that a self-organizing system, once defined in entirety, may create some sort of organizational hierarchy. To extend this hierarchy, it is necessary to modify in some way the units or interactions, otherwise by definition there is no way to change the system organization. The natural world's easily observable richness in levels of organization escapes these limitations through an incredibly complex interaction space and extremely long time scales. However, there is a flaw in this reasoning, in the assumption that a particular self-organizing system *must* reach some sort of organizational plateau. Though trivial, the example of aggregating triangles clearly demonstrates this type of behavior is a possibility (Dorin & McCormack, 2002). Less trivial examples may also be possible, but no examples of such are known to the author.

With the addition of this caveat, the ansatz seems quite reasonable, but limited in scope. Thermodynamics states that entropy increase everywhere is inevitable - *but only in a closed system*. Such systems are practically nonexistent (except perhaps the entirety of the universe). Similarly, fully-defined self-organizing systems are also closed, and as such are basically theoretical. In realistic models, external inputs must continually act on the components and environment (and are necessary if one wishes to model the *use* of such systems by humans). Just as in thermodynamics, it is of course possible to create systems which are closed in various degrees of approximation. The ansatz above tells us however that such closed systems will be uninteresting from a dynamic

organizational perspective, since they are *a priori* limited in scope. Instead, we must focus on how dynamic modification affects organizational hierarchies.

The CORAL model provides a single mechanism to perform these modifications: broadcast signals. Like many modular robotic studies, a *single* type of unit is used to build multiple topologies. Unlike these studies, the assembled unit *is not* re-programmed once the target topology is known, and instead units are immediately directed to assemble using the distributed background signals. While in edge cases it is probably possible to mimic re-programming CORAL units before assembly or to emulate manipulating each unit independently using broadcast signals, there is a pragmatic and theoretically interesting tradeoff where the unit itself is as simple as possible but still allows distributed control. Information is wonderfully mutable stuff, and this advantage can be better exploited when tied as little as possible to the hardware or wetware. In this case the behavioral information does not need to reside in any one unit, but in the structure itself, which leads naturally to scalable behavior (defined throughout the thesis in pragmatic computational terms). Since the structure, no matter how complex, is always designed for some sort of specified computation, extra steps determining how a composite structure acts are not necessary in order to compose meta-parts into larger structures. A shift in perspective is also required, where the developmental program being followed is externalized from the system itself, and only slowly built into the growing devices. As mentioned in Section 7.1 above, instead of a cellular metaphor for assembling devices, researchers investigating multi-scale assembly behavior may be better served by thinking of protein-based machinery. The parts themselves are much, much simpler, but from them entire cells can be built.

From a practical perspective, dynamic (re)organization in the sense described above may also be more interesting than pre-programmed self-organization since the former is the core ability that cannot be mimicked by existing machines designed for a single purpose. Instead of the question "can we build and program units which self-assemble into a device X?" one can ask "are there simpler units which can be the base of all devices?" A device is only useful if it has certain structure and can be controlled, where control has been historically the most difficult aspect of using systems assembled from many parts. Control comes naturally, however, when linked to the assembly model itself. In this thesis we have given simple examples of units universal in both form and computation, assembling into certain classes of controllable machines, modeled using two different computational paradigms and manipulated via external signals, while the machines themselves can take any tree or graph structure. Such units are blueprints for devices of arbitrary scale, which is major open problem of artificial life (Bedau et al., 2000; Lenaerts et al., 2005; Bedau, 2007). As with all computation, the behavior and structure of devices are fundamentally intertwined, though the underlying hope is that the potential for scalable assembly allows this requirement to be relaxed, as it is in our own artificial devices and indeed our own bodies.

Chapter 8

Conclusion

The core idea of assembly in materially closed but informationally open systems has served, in this thesis, as the basis for several interdependent investigations into dynamic, scalable assembly processes. These include the dynamic growth of logical devices, Petri net computation, scalable evolved assemblers, and a survey of the practical and theoretical implications of recursion in assembly. Collectively, this work addresses open problems in multi-level constructive dynamical systems, and demonstrates the novel capabilities of large numbers of extremely simple but composable devices. Given enough time and a carefully chosen basis, one can provably build arbitrary and nested complexity in a physically meaningful way.

In this chapter, results from previous chapters are integrated and summarized. We begin again with NOR units, as the simplest demonstrated example of a recursively assembled device of arbitrary scale and complexity. C/E unit and evolutionary results are next highlighted, demonstrating arbitrary scalable computation and the automatic generation of non-human-designed dynamically assembling systems. Finally, future experiments which would extend and further refine the research presented here are discussed, such as topological assembly restrictions, new genotype models, and co-assembly of device "brain" and "body."

8.1 NOR units

NOR units, by exploiting the inherent composability of NOR logical operations, are shown in this thesis to be capable of building controllable assembling structures of unlimited scale using only highly limited memory (36 bits). They are fundamentally limited in the types of computations the structures can produce and the way in which the inputs and outputs are sent, but even so the control of these units over arbitrary scales is well-defined and the structures built can take the form of any tree shape. Importantly, this is *not* simply a trivial consequence of the NOR operation. If units contained only a NOR operation encoded as a C/E net, the structures would behave identically but there would be no way to build or control them. This point is important and subtle: the developmental process itself must scale the same way as other functionality does.

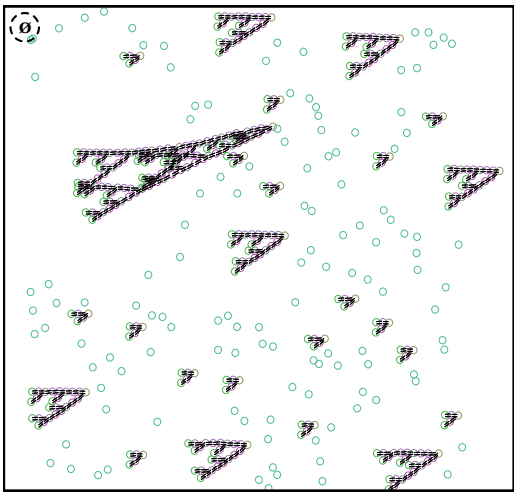


Figure 8.1: NOR units, demonstrating arbitrarily scalable assembly.

The novel demonstrated ability of NOR units to scale arbitrarily in simulation has important consequences for the design of real assembling devices. The CORAL environment in which the NOR units interact ensures no blatant shortcuts have been taken which make actual devices impractical: pairwise interactions, stochastic mixing, and broadcast signaling have been demonstrated in multiple ways for both machines and molecules (as further discussed in Chapters 2 and 3), though undoubtedly limitations remain for many domains. The relatively complex internal logic of NOR units makes the robotic implementation seemingly more realistic, though much of this logic is devoted to tasks that would be redundant in a chemical environment.

For example, because no *a priori* environmental shape or influences are assumed for a CORAL unit, the modification of behavior, even trivial, after connections are made requires explicit C/E net logic devoted to "remembering" the new structure. For many assembling systems, behavior when linked is implicitly modified by other environmental factors, such as other ports being blocked by the physical shape of a connected unit. NOR units must ignore environmental signals encoding logical inputs once connected to other units at the corresponding *In*-port, which requires several additional C/E places and transitions to track. If built as a real device, however, a reasonable optimization would be to place the sensor inside the recessed port itself. Depending on the unit and environment, this might be enough to block the sensor when the port is connected. Similarly, explicit clock signals may be unneeded if the internal transitions are implicitly timed more strictly. The lack of these assumptions in the CORAL model allows the computational contribution of these environmental properties to be measured quantitatively, by the amount of core C/E logic removed. While real systems might simplify the logic of a CORAL assembler this way, the hope is that as much as possible no new functions would need to be *added*. NOR units thus are a blueprint for how to build a controllable, scalable system in general.

8.2 Assembly of arbitrary computers

The second assembler design presented, C/E units, addresses the computational limitations of feed-forward Boolean logic. In contrast to a different model laid overtop the core C/E net units

of the underlying CORAL model, one can instead define an assembly model based on Petri nets themselves using a C/E net primitive. Through the composition of this C/E net with itself, any other safe Petri net of arbitrary size can be emulated. As discussed in Chapter 5, safe Petri nets are essentially equivalent in computational power to linear-bounded automata (Esparza, 1998). As the size of the network grows, the automata bound grows as well, so that a linear bounded automata with any size tape is theoretically possible to emulate. Thus an assembling system of C/E units is Turing complete to the same extent as any other realistic computer, i.e. the C/E primitive can be directed to assemble arbitrary computing devices via assembly. No disparate higher-level abstraction needs to be generated using these primitives in order to control these devices; the internal logic of a meta-C/E unit is realized using a composition of many other assembled C/E units.

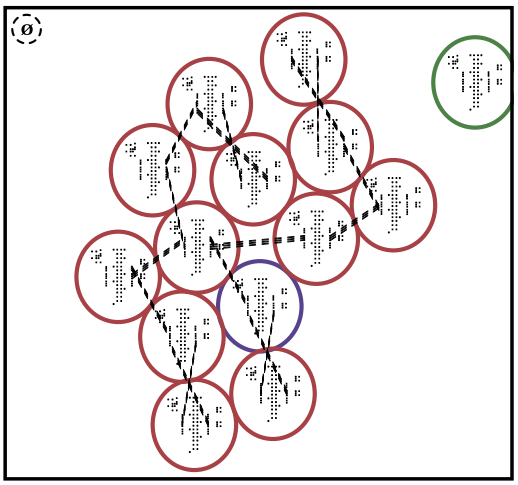


Figure 8.2: C/E units, configured via assembly to act as part of a larger C/E meta-net.

The lack of intermediate abstractions provides an interesting perspective on assembly which is not shared by NOR units. Assembled NOR unit structures and meta-structures might be thought of as a one-directional ray in assembly space: starting from the base atomic implementation (using C/E nets), meta-NOR units can be assembled at indefinitely higher and higher scales. Continuing this same analogy, C/E unit assembly would be a full line extending indefinitely in *both* directions. The presumably atomic C/E net base can in fact itself be decomposed into a construction of other C/E units, which themselves can be decomposed, *ad infinitum*, via the island graph construction from Section 5.5.1. There is therefore no inherent scale to C/E unit operations, though the assembly algorithm itself currently remains tied to only the atomic primitive.

8.2.1 Stochastic graph assembly

The introduction of C/E units in Chapter 5 also provides insights into the general construction of complex structures containing cyclic edges in a stochastic environment. Without cycles only tree structures are formed (by definition), each of which has a well-defined root where assembly may begin. Given the only active components in a CORAL environment are individual units

and the growing tree structures, there is no need to verify that a new element added to the tree is a particular unit - assembly proceeds deterministically. This is not the case when a structure with cycles is required. Because the CORAL model allows no way for units to influence the particular complementary port to which an open port will attach, structures requiring a cyclic edge between two sub-units must verify that they have not instead become attached in longer chains or multi-structure rings via *other* structures' identical sub-units (much more likely). Like unintended outputs in chemical reactions, these additional assembly products must be filtered until only the desired self-linked structure remains.

As described in Section 5.4.2, a stochastic method is required for this filtering. While the algorithm itself is not too complex, the process is perhaps best described by analogy. After assembly of a cyclic edge has occurred, resulting in self-connected, chain, and ring structures, one can imagine sending input which vigorously "shakes" these structures which are "weak" at the newly-created connections. If the shaking ever bends two connected sub-structures in opposite directions, the connection breaks. Self-connected units are stable because they have only a single sub-structure and so can only be bent in a single direction, but chains and rings eventually disintegrate. In the C/E unit implementation, shaking is represented by each connected sub-structure choosing to send a token (or not to send a token) backwards from the new cyclic link target unit, through the internal sub-structure, reaching a potentially different link target unit. If these units did not previously make the same token choice they must be different units, and the link is broken. The algorithm, as a part of the C/E "turtle" assembly process, is also applicable to other assembling models in stochastic, well-mixed environments. While other stochastic assembly algorithms for target structures are known (White et al., 2005; Grushin & Reggia, 2008), locality is assumed as a key factor, components are inflexible, and there are additional environmental assumptions.

8.3 Assembly and virtual evolution

To recap the discussion of Chapter 6, optimizing scalable assembling units using evolutionary algorithms seems to be a difficult task, even in the simple pairing case. This correlates with results from other researchers doing related experiments in evolved assembly, where the deterministic construction of simple group structures is non-trivial (Koza, 1992; Harvey & Thompson, 1996; Miller & Banzhaf, 2003; Roggen & Federici, 2004; Zykov et al., 2007), particularly if multi-scale artifacts or organizations are desired (Studer & Lipson, 2006). Overall the 1+1 netcrawler evolutionary algorithm was by far the most efficient choice for evolving recursive pairing units, though population-based methods were penalized heavily for time-consuming intermediate fitness evaluations. The best assemblers all exploited a similar stochastic recursive pairing behavior, which probably reflects the limitations of connectivity and internal unit logic assumed for the units. The optimal amount of logic (~5 places, ~8 transitions) for even this extremely simple recursive task can be compared with the designed logic capable of building complex recursive structure.

Finding *consistent* and *composable* behavior in the evolutionary search was challenging, and testing for either of these properties was difficult using a fitness function. Consistency requires many tests given an unpredictable environment, while also introducing multiple complicating factors related to the degree of penalty one wishes to assign to partial failure. Compositional behavior

is, by definition, reliant on previous behaviors, and so simple malfunctions at lower levels trickle upwards and destroy the ability to meaningfully evaluate the higher-level interactions.

One solution to the compositional problems may be methods which auto-group related behaviors together, either implicitly or explicitly, preferably across assembly scales. One major approach to this idea is to use developmental encodings (Bentley & Kumar, 1999; Kumar & Bentley, 2003; Stanley & Miikkulainen, 2003; Harding & Banzhaf, 2008). These methods are related to the grammatical encodings discussed previously for Petri net evolution (Moore & Hahn, 2004b) as well as developmental encodings for neural networks (Jakobi, 1995; Yao, 1999). Many developmental methods naturally generate phenotypes of many sizes, which is a useful feature when the optimal size of the solution is not known beforehand (as is the case with recursive assemblers). One must be careful, however, not to equate the evolved size of a CORAL unit's C/E controller with the size of the structures it produces - generally *smaller*, *simpler* controllers are better. Nevertheless, the repeated behavioral patterns necessary for scalable assembly may be reflected in an easier-to-evolve form using developmental encodings which themselves exploit recursion and modularity.

More tests, and therefore computational power, can always solve the consistency problems, though there may be a more elegant approach if the assembling system can be approximated in more detail. As discussed previously in Section 2.4.2, approximation methods for stochastic assembling systems have been developed, and these could be adapted to the CORAL model in the future. Depending on the assumptions made about the interactions and environment, the methods are not necessarily faster given a constructive system, e.g. (Dittrich et al., 2001), but there are undoubtedly many simple cases which can be assigned a meaningful value quickly. On the other hand, formal methods for characterizing the *internal* behavior of CORAL units could also be used, though analytic methods would probably be hard to apply across structure growth in cases where the environment is not mostly stochastic. In general, fully formalizing the pre-configuration assembly problem for Petri net CORAL units has not been successful to date but is definite goal of future work.

Though artificial evolution bears little resemblance to natural evolution in many ways, there have been partially-successful attempts to generate multiple levels of structural organization in virtual evolved systems (Studer & Lipson, 2006), mimicking the hierarchical organization seen in natural systems. As mentioned in Chapter 2, other work has theorized that hierarchical organization and "open-ended" evolution may require either new types of interaction or extremely distributed computational processing. The experiments described in this thesis support the latter view, since the generation of structure and function across scales was demonstrably not limited by the simple pairing interactions used by the CORAL model. Arbitrarily hierarchical organization is possible in the CORAL model, as is the deterministic control over any level of this organization, though when artificially evolving such systems the precision required is difficult to achieve using traditional techniques. It is an indulgent hypothesis of the author that the many natural hierarchies inherent in the biological world may partially result from evolutionary reorganization under constant physical, geological, and cosmic perturbations.

8.4 Future work

While limitations and particularly interesting areas for further study have been noted throughout the text, some of the main areas in which further research may lead to particularly interesting results are again mentioned here. Probably the most important of these is the development of an explicit mechanism to combine structures created at different scales. It is possible using NOR units to build intricate logical structures as well as large meta-components. Assembling the two together has not been demonstrated, but would be a significant breakthrough. One might imagine the "brain" being assembled first, as an intricate small-scale structure with many components, then temporarily disabling itself and mimicking a structural meta-unit. After meta-unit assembly of the larger body, the brain structure can be re-enabled, resulting in a fully functional and intelligent device built from only a single part, using a methodology applicable in a range of environments. This ability may not even require extra scaffolding constructs, if the brain structure is designed such that a subset of its function is the same as structural units.

While NOR-units provide the recursive assembly base from which to attempt the ideas above, the potential devices would be more capable if C/E units were also extended with recursive assembly ability as well. The direct emulation of a meta-C/E controller solves most of the computation issues, but there still must be a way in which to disable internal units' response to background signals. This might be done explicitly, through the use of turtle state modification and a new signal, but there may also be a more elegant way exploiting the C/E meta-structure itself.

As a primary goal of the research presented here is the demonstration of a realistic assembler, one of the other major unaddressed issues is the lack of a model for locality and space. There are reasons for the omission, since any particular type of environmental assumption makes the computational unit less general. Certain interesting properties may emerge from a minimal lattice model, however. For example, C/E units require six ports, which might be mapped onto a cube with six faces. Given this topological restriction, is it still possible to create arbitrary computing devices? The author believes that it would be possible, but the question is open. Even if the restriction does not affect computability, it would be an interesting investigation to determine the efficiency of various packings in terms of units required. Assemblers with unconstrained topology, as presented in this thesis, provide a pragmatic and meaningful benchmark against which to judge assembly in different spaces.

The evolutionary search for scalable assembling units has only begun, though it has been seen that such a search is effectively genetic programming and therefore quite challenging. The multi-scale tests required to evaluate evolved units make it even more so. To date, only single signals are used by evolving units. A coevolutionary approach may better represent the problem, where a limited assembly algorithm generating multiple types of signals evolves alongside the assembling units themselves. Developmental encodings may also prove useful. Multi-level fitness evaluation is still problematic, but an analytic approach in which the unit controllers are not simulated directly (a topic touched on briefly in the filtering of inactive or non-responsive units) may show promise. Markov chain analysis may be possible with different timing assumptions for CORAL units, even if used only as a first approximation to behavior. In particular, exploring how different timing mechanisms could enhance realism and analysis in CORAL units is necessary for further study from both a scientific and engineering point of view.

8.5 Final remarks

The results and main contributions of this thesis have been related in the sections above, but are restated here in summary. They include:

- a novel design for a recursively assembling device capable of building controllable structure at arbitrary scale,
- a novel design for a device able to assemble into any other computing device, without pre-programming,
- the CORAL model, a minimal framework for computational assembly respecting conservation of mass, which ensures the general applicability of the above prototypes,
- and a search for other interesting instances of such devices using evolutionary algorithms, with a comparative study of different types of evolutionary search.

The prime motivation for this thesis can be expressed in even more fundamental terms as a search for the simplest interface between information and the physical world. How thinly can we slice our machines, artificial and biological, before we lose essential control and are left with only sand? Aside from the practicality of such a minimal unit, the answer to this question serves as a baseline or metric from which to understand the causally-connected interplay between a structure and how it behaves. The work of this thesis demonstrates clearly that such an interface is possible, has useful properties, and is well within our current technical abilities, but after this experience it seems likely that even more elegant, minimal, and useful "slices" can be developed with further work. As has been hypothesized by many, the linking of the informational world of the 20th century to the new physical, chemical, and biological world of the 21st will begin another revolution in science and society. This work is a small contribution toward that lofty goal, through the creation of a computational bridge between structure and function at arbitrary scale.

Appendix A

Proof of meta-unit shielding

In this appendix, the ideas presented earlier in Section 4.3.2 regarding NOR meta-unit shielding behavior are more rigorously treated. While a partial example is given in Figures 4.10 and 4.11, NOR meta-unit shielding must be shown to hold for larger meta-units so that the waterfall selection and assembly algorithm can be confidently said to assemble meta-units of arbitrary size.

Following a similar structure to the blinking proof of Section 4.3.1, first some definitions of NOR meta-units, repeating signals, and shielding are presented. These definitions are then used in a recursive proof, which defines the shielding property starting from a meta-unit of individual units.

Definition 4. A repeated background signal of A is defined as a series of repeating signal groups of the form $ACK\ ACK\ ACK\ ACK\ ACK\ ACK\ldots$, consisting of repeated groups of A and (clock signals). Repeated signals of B are defined similarly.

Definition 5. A repeated signal of length n is defined as a repeated signal consisting of n signal groups of ACK or BCK.

Definition 6. A NOR meta-unit of scale $s \in \mathbb{N}_0$ is defined as follows:

- A NOR meta-unit of scale $s = 0$ is an individual NOR unit.
- A NOR meta-unit of scale $s > 0$ is a structure built from NOR meta-units of scale $s - 1$, also called *sub-units*. The ports of the sub-units are linked as in Figure A.1, and the NOR meta-unit always has a single *Out*-port and two *In*-ports as defined in Section 4.3.2. Figure A.1 also establishes numeric labels for each sub-unit.

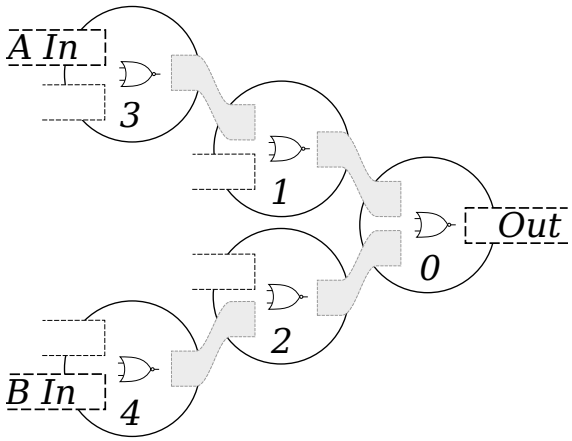


Figure A.1: NOR meta-unit of connected sub-units (scale > 0). The labeled *In*-ports and *Out*-port are the only ports which may be connected to the complimentary ports of other meta-units.

As can be seen in the diagram, the *Out*-port and *In*-ports of a meta-unit are also the *Out*-port of the root sub-unit and the topmost and bottommost *In*-port of sub-units 3 and 4. The *output* of a meta-unit is also defined as the output of its root sub-unit when the scale > 0.

Definition 7. A NOR meta-unit of scale $s > 0$ *shields* its *In*-ports from a repeating *A* or *B* signal of length 3^s when:

- If connected at the corresponding *A* or *B In*-port with input x from the connected unit prior to the repeated signal, the (root) output is $\neg x$ after the repeated signal.
- If open at the corresponding *A* or *B In*-port, the output after the repeated signal is the same as if the port was connected and $x = true$ prior to the repeated signal. The output will therefore always be *false*. Essentially the repeated signal emulates sending a *true* input to the corresponding port in the same way an individual *ACK* or *BCK* signal group does for individual NOR units.

It is important to note that individual NOR units of scale $s = 0$ do *not* shield their inputs by this definition. Even when receiving an *A* or *B* signal, individual units also take input from the opposite connected *In*-port into account, leading to a full NOR'ed output after the clock signal. Meta-units are capable of only a weaker shielding of one input or the other, though this half-protection is enough for recursive assembly. The length of the repeating signal is also important. It may be possible to relax this assumption somewhat, using more or less signals, but only a single level of meta-unit shielding is ever necessary for the waterfall assembly algorithm and so the more general case is not addressed.

Theorem 8. NOR meta-units of scale $s > 0$ shield their *In*-ports from a repeated signal of length 3^s.

Case 1. *NOR meta-units of scale* 1 *shield their* In*-ports from a repeated signal of length* 3^1.

A NOR meta-unit of scale 1 is, by definition, a connected structure of five individual NOR units. We assume a repeating *A* signal, and claim by symmetry the same argument holds for repeating *B* signals.

After any *ACK* signal group, the output of sub-units 2 and 4 will always be false, by the operation of individual NOR units. To explain further, sub-units 2 and 4 have open *A In*-ports, and so they interpret an *A* signal as a true value for one of the variables in their calculated output (as discussed in Section 4.2). The output of these NOR units must then always be $true \downarrow x = false$, where x is any Boolean input from the In-port connected unit.

Conversely, under the influence of a repeating *A* signal, the open *B In*-ports of sub-units 1 and 3 always ensure one false input. The output of sub-units 1 and 3 therefore depends entirely on the respective input from each sub-unit's connected *In*-port. After the first *ACK* signal group, the output of sub-unit 3 will be $x \downarrow false = \neg x$, where x is either *true* (if not connected at the *A In*-port) or is the input value of the connected unit before the repeated *A* signals. Again, this holds by the operation of individual NOR units, described in Section 4.2.

This output becomes the input for sub-unit 1, since sub-unit 3's *Out*-port is connected to sub-unit 1's *In*-port. After a second *ACK* signal group, the output for sub-unit 1 (and *A In*-port input for sub-unit 0) will be $(\neg x) \downarrow false = \neg(\neg x) = x$. Given the above-mentioned property that the output of sub-unit 2 is always false after any *ACK* signal group, the output of the root sub-unit 0 depends entirely on sub-unit 1. After the third signal group, this root output will be $x \downarrow false = \neg x$.

Since the output of root sub-unit 0 and therefore (by definition) the NOR meta-unit is $\neg x$ after a repeated signal of length 3, where x is either *true* (if not connected at the *A In*-port) or the input value of the connected meta-unit, the NOR meta-unit shields its *A In*-port from a repeated *A* signal (of length 3). The structure and ports of a NOR meta-unit are symmetric, and so the same argument holds for shielding of the *B In*-port from a repeated *B* signal.

Case 1. *Assuming the theorem holds for NOR meta-units of scale* $s \geq 1$*, NOR meta-units of scale* $s+1$ *shield their* In*-ports from a repeated signal of length* 3^{s+1}.

The proof proceeds largely along the same lines as the previous case. A NOR meta-unit of scale $s+1$ is, by definition, a connected structure of five sub-units of scale s, labeled as in Figure A.1. We again assume a repeating *A* signal, and claim by symmetry the same argument holds for repeating *B* signals.

By the recursive assumption, all sub-units shield their *A In*-ports from a repeating *A* signal of length 3^s. After 3^s repetitions of signal groups, therefore, the output of sub-units 2 and 4 (Figure A.1) will always be false. We can also derive immediately that the output of sub-units 0 and 1 will always be the inverse of the *A In*-port input 3^s repetitions ago. Chaining these ideas together, the output of sub-unit 0 will always be the *same* as the *A In*-port input to sub-unit 1 after 2×3^s repetitions.

Again by the recursive assumption, we know sub-unit 3 shields its *In*-ports from a repeated signal of length 3^s. The output of sub-unit 3 after 3^s repetitions is therefore $\neg x$, which is also the input to sub-unit 1. By the derivation in the previous paragraph, after an additional 2×3^s repetitions the output of sub-unit 0 will be identical to the output of sub-unit 3: $\neg x$. Since x is either the input from the connected unit at the meta-unit *A In*-port prior to the $3^s + 2 \times 3^s = 3^{s+1}$ repeated signals or is *true*, and the root output of the meta-unit of scale $s+1$ after the repeated signal is $\neg x$, NOR meta-units of scale $s+1$ also shield their *A In*-ports from a repeated *A* signal. Symmetrically, the same holds for the *B In*-port and repeated *B* signals.

Proof. Since NOR meta-units of scale $s = 1$ *shield their* In*-ports from a repeated signal of length*

3^s*, and this property also holds for NOR meta-units of scale* $s+1$ *when it holds for scale s, NOR meta-units of any scale* $s > 0$ *shield their* In*-ports from a repeated signal of length* 3^s. □

Intuitively, the above theorem shows how one can select between structures of many connected meta-units and single meta-units. Multi-meta-unit structures will always have a root connected to further child meta-units, and the presence of this connection changes the output as compared to a "free" or individual meta-unit in response to particular repeated signals. The core of the waterfall assembly algorithm relies on the ability to make this distinction.

Appendix B

A comparison of C/E primitives

Section 5.2 introduces a compositional net primitive developed by Petri while attempting to model regular computational structures which obey physical and relativistic metrics. In doing so, Petri found that a certain type of "noisy channel" net was capable, in combination, of performing arbitrary, reversible, Boolean computation (Petri, 1996). Though the motivation comes from a computational basis of physics, not building assembling devices, such a primitive provides an interesting comparison to the C/E net primitive of Chapter 5.

Because of the different research focus, the noisy channel primitive was not designed for interactive computation, and more flexible methods of composition are required. However, the construction itself is quite elegant, shown below as Figure B.1a, and with minor additions can be transformed into a version compatible with the CORAL model.

As shown in Figure B.1, merging transitions allows four copies of the "noisy channel" primitive to emulate the Boolean Quine function:

$$Q(a,b,c) = a\bar{b} \vee bc$$

This function, when composed with itself, can emulate any other Boolean switching network (Petri & Smith, 1987), and Boolean switching networks are the core of computational technology used in many areas. However, in order to perform this composition it is necessary to merge *places* as well as transitions, sharing state *between* merged nets. Without place merging, the only types of places in networks composed of multiple noisy channel primitives are those with solely outgoing edges and those with solely incoming edges. This forces deadlock - places with solely incoming edges never enable further transitions for firing and there is no way for new tokens to enter places with outgoing edges. In order to allow more interactive behavior, place merging can be emulated by the transition merging allowed in the CORAL model, attaching an additional outgoing or incoming transition to each place. Technically each transition must also be assigned a complimentary "direction," but this can be done such that the Quine construction is still possible. Figure B.2 shows the result: a compositional unit rather similar to the C/E net primitive of Figure 5.1.

The final noisy channel primitive requires at least 8 controllable port transitions to form Quine

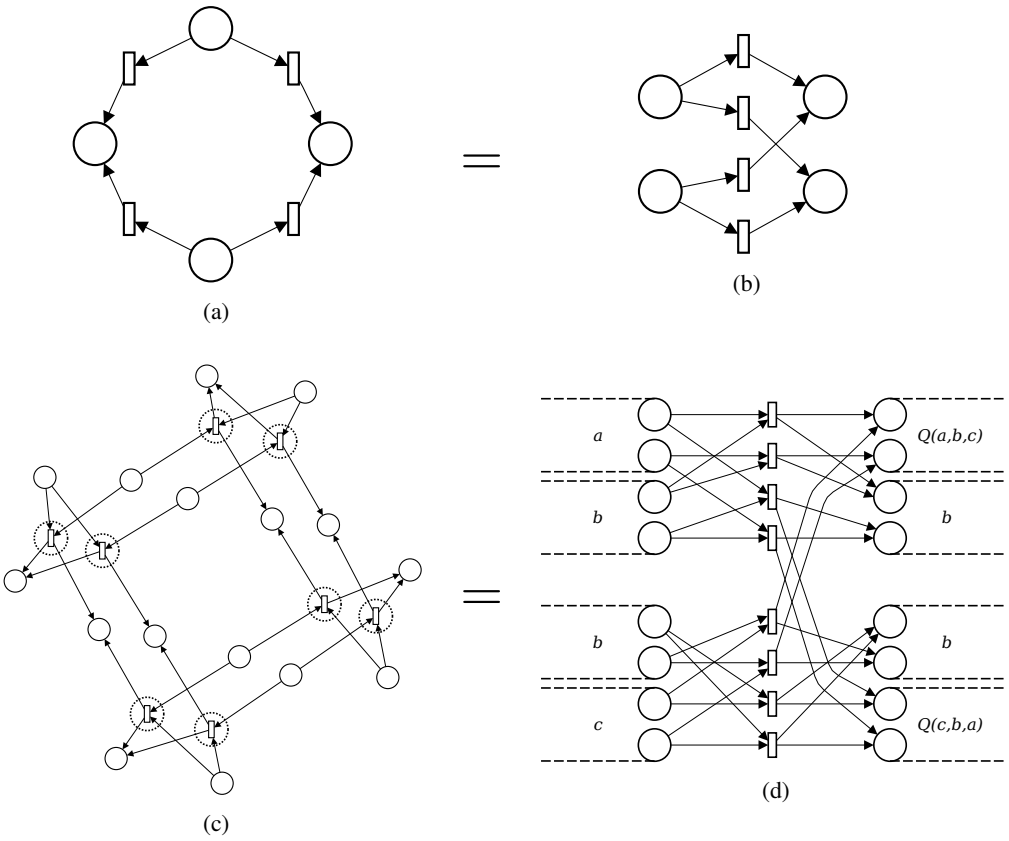


Figure B.1: Base construct and the composed quine function from (Petri & Smith, 1987; Petri & Reisig, 2008). Subfigures (B.1a) and (B.1b) are the "noisy channel" primitive, which can be composed by combining transition edges into the computationally universal Quine function of (B.1c) and (B.1d). The dotted circles in (B.1c) indicate transition mergings. In (B.1d) the places are grouped in high/low pairs for each bit of the Quine transfer function. To compose the construction of (B.1d) with any other, assuming only pairwise interactions (as in the CORAL model), additional transitions would be needed to link the incoming and outgoing places.

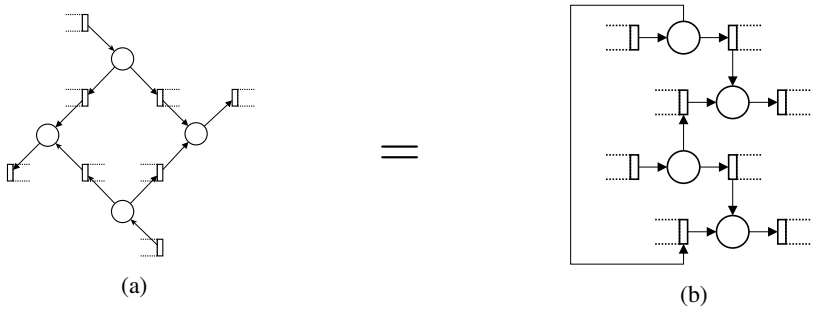


Figure B.2: The noisy channel primitive of Figure B.1 with additional input and output transitions needed by the CORAL model. A redrawn version is shown as (B.2b), making the similarity with the C/E net primitive of Figure 5.1 clear.

functions, and 4 places to store state. This is two more ports than are required for the C/E net primitive of Chapter 5, and an additional place, though the structure itself is actually quite similar to the C/E net primitive. Without the additional transitions, the noisy channel primitive would be slightly simpler than the C/E net primitive, though there is still a need for controllable composition of the places - requiring a full eight ports of some kind. As discussed in Chapter 5, the control of ports is complex, so minimizing the port number may result in simpler assembly behavior than only reducing internal state. Given that the C/E net primitive is also, in compositions, computationally universal (the Quine function of Figure B.2b can be built using the island graph construction), it is a slightly simpler primitive in the CORAL context, though with a less regular topology.

Bibliography

Abelson, H., Allen, D., Coore, D., Hanson, C., Homsy, G., Knight, J., Nagpal, R., Rauch, E., Sussman, G. J., & Weiss, R. (2000). Amorphous computing. *Communications of the ACM*, *43*(5), 74–82.

Adleman, L. (1994). Molecular computation of solutions to combinatorial problems. *Science*, *266*(11), 1021–1024.

Adleman, L. (2000). Toward a mathematical theory of self-assembly. Tech. Rep. 00-722, University of Southern California, Department of Computer Science.
URL `http://www.usc.edu/dept/cs/tech.html`

Andeen, G. (1997). Toward a science of assembly. *Robotics and Autonomous Systems*, *21*, 239–248.

Ashby, W. R. (1962). Principles of the self-organizing system. In H. Foerster, & G. J. Zopf (Eds.) *Principles of Self-Organization*, (pp. 255–278). Pergamon Press.

Ashley-Rollman, M. P., Lee, P., Goldstein, S., Pillai, P., & Campbell, J. (2009). A language for large ensembles of independently executing nodes. In P. Hill, & D. Warren (Eds.) *Proceedings of the International Conference on Logic Programming (ICLP '09)*, vol. 5649 of *Lecture Notes in Computer Science*, (pp. 265–280). Pasadena, CA, USA: Springer.

Baas, N. (1994). Emergence, hierarchies, and hyperstructures. In C. Langton (Ed.) *Artificial Life III*, Santa Fe Institute Studies in the Sciences of Complexity, (pp. 515–537). Addison-Wesley.

Baas, N., Ehresmann, A., & Vanbremeersch, J. (2004). Hyperstructures and memory evolutive systems. *International Journal of General Systems*, *33*(5), 553–568.

Baas, N., & Helvik, T. (2005). Higher order cellular automata. *Advances in Complex Systems*, *8*(2-3), 169–192.

Bagley, R., Farmer, J., & Fontana, W. (1991). Evolution of a metabolism. In *Artificial Life II*, Santa Fe Institute Studies in the Sciences of Complexity, (pp. 141–158). Westview Press.

Bahçeci, E., Soysal, O., & Şahin, E. (2003). A review: pattern formation and adaptation in multi-robot systems. Tech. Rep. CMU-RI-TR-03-43, Carnegie Mellon University, Robotics Institute.
URL `http://www.ri.cmu.edu/publication_view.html?pub_id=4534`

Baldassarre, G., Nolfi, S., & Parisi, D. (2003). Evolving mobile robots able to display collective behaviors. *Artificial Life*, *9*(3), 255–267.

Banks, E. R. (1971). *Proof of universal computation in cellular automata.* Ph.D. thesis, Massachusetts Institute of Technology, Department of Mechanical Engineering.

Banzhaf, W. (2004). Artificial chemistries - towards constructive dynamical systems. *Solid State Phenomena*, *97*, 43–50.

Banzhaf, W., Dittrich, P., & Eller, B. (1999). Self-organization in a system of binary strings with topological interactions. *Physica D*, *125*, 85–104.

Banzhaf, W., Guillaume, B., Christensen, S., Foster, J., François, K., Lefort, V., Miller, J., Miroslav, R., & Ramsden, J. (2006). From artificial evolution to computational evolution: a research agenda. *Nature Reviews: Genetics*, *7*, 729–735.

Banzhaf, W., Nordin, P., Keller, R., & Francone, F. (1998). *Genetic Programming: An Introduction*. Morgan Kauffman.

Barnett, L. (2001). Netcrawling - optimal evolutionary search with neutral networks. In *Proceedings of the 2001 Congress on Evolutionary Computation (CEC)*, (pp. 30—37). Seoul, Korea: IEEE Press.

Barto, A., & Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, *13*(4), 341–379.

Bayindir, L., & Şahin, E. (2007). A review of studies in swarm robotics. *Turkish Journal of Electrical Engineering and Computer Science*, *15*(2), 115–147.

Beal, J. (2005). Amorphous medium language. In *AAMAS Large-Scale Multi-Agent Systems Workshop (LSMAS)*. Utrecht University.
URL `http://groups.csail.mit.edu/mac/projects/amorphous/papers/lsmas-final.pdf`

Bedau, M. (2007). Artificial life. In M. Matten, & C. Stephens (Eds.) *Philosophy of Biology*, vol. 3 of *Handbook of the Philosophy of Science*, (pp. 585–603). Amsterdam: Elsevier.

Bedau, M. (2009). Evolution of complexity. In *Mapping the Future of Biology: Evolving Concepts and Theories*, (pp. 111–130). Netherlands: Springer.

Bedau, M., McCaskill, J., Packard, N., Rasmussen, S., Adami, C., Green, D., Ikegami, T., Kaneko, K., & Ray, T. (2000). Open problems in artificial life. *Artificial Life*, *6*(4), 363–376.

Bedau, M., Snyder, E., Brown, C., Husbands, P., & Harvey, I. (1997). A comparison of evolutionary activity in artificial evolving systems and the biosphere. In *Proceedings of the Fourth European Conference on Artificial Life (ECAL)*, (pp. 125–134). Brighton, UK: MIT Press.

Benkö, G., Centler, F., Dittrich, P., Flamm, C., Stadler, B., & Stadler, P. (2009). A topological approach to chemical organizations. *Artificial Life*, *15*(1), 71–88.

Bentley, P., & Kumar, S. (1999). Three ways to grow designs: A comparison of embryogenies for an evolutionary design problem. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, vol. 1, (pp. 35–43). Orlando, FL, USA: Morgan Kauffman.

Bernardini, F., Brijder, R., Rozenberg, G., & Zandron, C. (2007). Multiset-based self-assembly of graphs. *Fundamenta Informaticae*, *75*(1), 49–75.

Bernardini, F., Gheorghe, M., Krasnogor, N., & Giavitto, J. (2005). On self-assembly in population P systems. In *Unconventional Computation*, vol. 3699 of *Lecture Notes in Computer Science*, (pp. 46–57). Sevilla, Spain: Springer.

Beyer, W., Sellers, P., & Waterman, M. (1985). Stanislaw M. Ulam's contributions to theoretical theory. *Letters in Mathematical Physics*, *10*, 231–242.

Bird, J., & DiPaolo, E. (2008). Gordon Pask and his maverick machines. In P. Husbands, M. Wheeler, & O. Holland (Eds.) *The Mechanization of Mind in History*, (pp. 185–212). MIT Press.

Bishop, J., Burden, S., Klavins, E., Kreisberg, R., Malone, W., Napp, N., & Nguyen, T. (2005). Self-organizing programmable parts. In *International Conference on Intelligent Robots and Systems (IROS)*, (pp. 2644–2651). Edmonton, Canada: IEEE Press.

Bonabeau, E., Guérin, S., Snyers, D., Kuntz, P., & Theraulaz, G. (2000). Three-dimensional architectures grown by simple 'stigmergic' agents. *Biosystems*, *56*(1), 13–32.

Bongard, J. (2008). Behavior chaining: incremental behavior integration for evolutionary robotics. In *Artificial Life XI*, (pp. 64–71). Winchester, UK: MIT Press.

Bongard, J. (2009). Biologically inspired computing. *IEEE Computer Magazine*, *42*(4), 95–98.

Bourdeaud'huy, T., & Yim, P. (2002). Petri net controller synthesis using genetic search. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, vol. 1, (pp. 528 – 533). Hammamet, Tunisia: IEEE Press.

Bremermann, H. (1962). Optimization through evolution and recombination. In M. Yovits, G. Goldstein, & G. Jacobi (Eds.) *Self-organizing systems*, (pp. 93–106). Washington, DC: Spartan.

Brooks, R., & Stein, L. (1994). Building brains for bodies. *Autonomous Robots*, *1*(1), 7–25.

Buchanan, A., Gazzola, G., & Bedau, M. (2008). Evolutionary design of a model of self-assembling chemical structures. In N. Krasnogor, S. Gustafson, D. Pelta, & J. Verdegay (Eds.) *Systems Self-Assembly: Multidisciplinary Snapshots*, (pp. 79–100). Elsevier.

Cao, Y., Fukunuga, A., & Kahng, A. (1997). Cooperative mobile robotics: antecedents and directions. *Autonomous Robots*, *4*(1), 7–27.

Cariani, P. (1993). To evolve an ear. epistemological implications of Gordon Pask's electrochemical devices. *Systems Research*, *10*(3), 19–33.

Casjens, S., & King, J. (1975). Virus assembly. *Annual Review of Biochemistry*, *44*, 555–611.

Castano, A., Behar, A., & Will, P. M. (2002). The Conro modules for reconfigurable robots. *IEEE/ASME Transactions on Mechatronics*, *7*(4), 403–409.

Chen, I., & Burdick, J. (1995). Determining task optimal modular robot assembly configurations. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, (pp. 132–137). Nagoya, Japan: IEEE Press.

Cheng, J., Cheng, W., & Nagpal, R. (2005). Robust and self-repairing formation control for swarms of mobile agents. In M. Veloso, & S. Kambhampati (Eds.) *Proceedings of the Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference*, vol. 20, (pp. 59–64). Pittsburgh, USA: AAAI Press / MIT Press.

Chiu, Y., & Fu, L. (1997). A GA embedded dynamic search algorithm over a Petri net model for an FMS scheduling. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, (pp. 513–519). Albuquerque, New Mexico: IEEE Press.

Chou, H., & Reggia, J. (1997). Emergence of self-replicating structures in a cellular automata space. *Physica D: Nonlinear Phenomenon*, *110*, 252–276.

Clark, A. (2008). *Supersizing the Mind*. Oxford University Press.

Codd, E. (1968). *Cellular Automata*. New York: Academic Press.

Corradini, A. (1995). Concurrent computing: from Petri nets to graph grammars. In *Proceedings of the Joint COMPUGRAPH/SEMAGRAPH (SEGRAGRA) Workshop on Graph Rewriting and Computation*, Electronic Notes in Theoretical Computer Science, (pp. 56–70). Pisa, Italy: Elsevier.

Costelha, H., & Lima, P. (2008). Modeling, analysis and execution of multi-robot tasks using Petri nets. In *Proceedings of the Seventh International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, (pp. 1187–1190). Estoril, Portugal: International Foundation for Autonomous Agents and Multiagent Systems.

Crutchfield, J., & Görnerup, O. (2006). Objects that make objects: the population dynamics of structural complexity. *Journal of the Royal Society Interface*, *3*(7), 345–349.

Crutchfield, J., & Young, K. (1989). Inferring statistical complexity. *Physical Review Letters*, *63*(2), 105–108.

Danos, V., Feret, J., Fontana, W., Harmer, R., & Krivine, J. (2008). Rule-based modeling, symmetries, refinements. *Formal Methods in Systems Biology*, *5054*, 103–122.

Danos, V., Feret, J., Fontana, W., & Krivine, J. (2007). Scalable simulation of cellular signaling networks. In S. Zhong (Ed.) *Asian Symposium on Programming Languages and Systems (APLAS)*, vol. 4807 of *Lecture Notes in Computer Science*, (pp. 139–157). Singapore: Springer.

Danos, V., Krivine, J., & Tarissan, F. (2006). Self-assembling trees. In *Proceedings of the Third Workshop on Structural Operational Semantics (SOS)*, vol. 175(1) of *Electronic Notes in Theoretical Computer Science (ENTCS)*, (pp. 19–32).

Danos, V., & Laneve, C. (2004). Formal molecular biology. *Theoretical Computer Science*, *325*(1), 69–110.

Danos, V., & Tarissan, F. (2007). Self-assembling graphs. *Natural Computing*, *6*(3), 339–358.

Detweiler, C., Vona, M., Yoon, Y., Yun, S., & Rus, D. (2007). Self-assembling mobile linkages. *IEEE Robotics & Automation Magazine*, *14*(3), 45–56.

di Fenizio, P., Dittrich, P., Banzhaf, W., & Ziegler, J. (2000). Towards a theory of organizations. In *German Workshop on Artificial Life (GWAL)*. Bayreuth, Germany: not in print.
URL `http://web.cs.mun.ca/~banzhaf/papers/gwal2000.pdf`

DiCesare, F. (1993). *Practice of Petri nets in manufacturing*. Chapman & Hall.

Dittrich, P., & Banzhaf, W. (1998). Self-evolution in a constructive binary string system. *Artificial Life*, *4*(2), 203–220.

Dittrich, P., & di Fenizio, P. (2007). Chemical organisation theory. *Bulletin of Mathematical Biology*, *69*(4), 1199–1231.

Dittrich, P., Ziegler, J., & Banzhaf, W. (2001). Artificial chemistries - a review. *Artificial Life*, *7*(3), 225–275.

Dorf, R. (1990). *Concise International Encyclopedia of Robotics: Applications and Automation*. Wiley-Interscience.

Dorigo, M., & Colombetti, M. (1994). Robot shaping: Developing situated agents through learning. *Artificial Intelligence*, *70*(2), 321–370.

Dorin, A. (2000). Creating a physically-based, virtual-metabolism with solid cellular automata. In M. Bedau, J. McCaskill, & S. Rasmussen (Eds.) *Artificial Life VII*, (pp. 13–20). Portland, Oregon: MIT Press.

Dorin, A., & McCormack, J. (2002). Self-assembling dynamical hierarchies. In *Artificial Life VIII*, (pp. 423–428). Sydney, Australia: MIT Press.

Doursat, R. (2008). Organically grown architectures: creating decentralized, autonomous systems by embryomorphic engineering. In R. Wuertz (Ed.) *Organic Computing*, Complex Systems Series, (pp. 201–220). Springer.

Drexler, K. E., Randall, J., Corchnoy, S., Kawczak, A., & Steve, M. (Eds.) (2007). *Productive nanosystems: a technology roadmap*. U.S. Department of Energy.
URL `http://e-drexler.com/d/07/00/1204TechnologyRoadmap.html`

Dudek, G., Jenkin, M., Milios, E., & Wilkes, D. (1996). A taxonomy for multi-agent robotics. *Autonomous Robots*, *3*(4), 375–397.

Egri-Nagy, A., & Nehaniv, C. (2004). Algebraic hierarchical decomposition of finite state automata: comparison of implementations for Krohn-Rhodes theory. In M. Domaratzki, A. Okhotin, A. Salomaa, & S. Yu (Eds.) *International Conference on Implementation and Application of Automata (CIAA)*, Lecture Notes in Computer Science, (pp. 315–316). Kingston, Canada: Springer.

Egri-Nagy, A., & Nehaniv, C. L. (2008). Hierarchical coordinate systems for understanding complexity and its evolution, with applications to genetic regulatory networks. *Artificial Life*, *14*(3), 299–312.

Eigen, M., & Schuster, P. (1977). A principle of natural self-organization. *Naturwissenschaften*, *64*(11), 541–565.

Esparza, J. (1998). Decidability and complexity of Petri net problems - an introduction. In W. Reisig, & G. Rozenberg (Eds.) *Lectures on Petri Nets I: Basic Models*, no. 1491 in Lecture Notes in Computer Science, (pp. 374–428). Springer.

Esparza, J., & Nielsen, M. (1994). Decidability issues for Petri nets - a survey. Basic Research in Computer Science (BRICS) RS-94-8, University of Aarhus, Denmark.
URL `http://www.brics.dk/RS/94/8/`

Esparza, J., Römer, S., & Vogler, W. (2002). An improvement of McMillan's unfolding algorithm. *Formal Methods in System Design*, *20*(3), 285–310.

Ewaschuk, R., & Turney, P. D. (2006). Self-replication and self-assembly for manufacturing. *Artificial Life*, *12*(3), 411–433.

Faeder, J., Blinov, M., Goldstein, B., & Hlavacek, W. (2005). Rule-based modeling of biochemical networks. *Complexity*, *10*(4), 22–41.

Feynman, R. (1959). There's plenty of room at the bottom. In *American Physical Society*, as presentation. California Institute of Technology, CA, USA.
URL `http://www.zyvex.com/nanotech/feynman.html`

Flocchine, P., Prencipe, G., Santoro, N., & Widmayer, P. (2008). Arbitrary pattern formation by asynchronous, anonymous, oblivious robots. *Theoretical Computer Science*, *407*(1-3), 412–447.

Fogel, D. (1998). *Evolutionary Computation: The Fossil Record*. Wiley-IEEE Press.

Fogel, L., Owens, A., & Walsh, M. (1965). Artificial intelligence through a simulation of evolution. In M. Maxfield, A. Callahan, & L. Fogel (Eds.) *Biophysics and Cybernetic Systems*, (pp. 131–155). Spartan.

Fontana, W. (1992). Algorithmic chemistry. In C. Langton, C. Taylor, J. Farmer, & S. Rasmussen (Eds.) *Artificial Life II*, Santa Fe Institute Studies in the Sciences of Complexity, (pp. 159–209). Westview Press.

Fontana, W. (2006). Pulling strings. *Science*, *314*(5805), 1552.

Fontana, W., & Buss, L. (1994). Arrival of the fittest: toward a theory of biological organization. *Bulletin of Mathematical Biology*, *56*(1), 1–64.

Fontana, W., & Buss, L. (1996). The barrier of objects: from dynamical systems to bounded organizations. Working Paper WP-96-27, International Institute for Applied Systems Analysis, Laxenburg, Austria.

Fredslund, J., & Mataric, M. (2002). A general algorithm for robot formations using local sensing and minimal communication. *IEEE Transactions on Robotics and Automation*, *18*(5), 837–846.

Freitas, R. (1980). A self-reproducing interstellar probe. *Journal of the British Interplanetary Society*, *33*, 251–264.

Freitas, R. J., & Merkle, R. (2004). *Kinematic Self-Replicating Machines*. Landes Bioscience. URL `http://www.molecularassembler.com/KSRM.htm`

Fukuda, T., Buss, M., Hosokai, H., & Kawauchi, Y. (1991). Cell structured robotic system CEBOT: control, planning, and communication methods. *Robotics and Autonomous Systems*, *7*(2-3), 239–248.

Fukuda, T., & Nakagawa, S. (1987). A dynamically reconfigurable robotic system (concept of a system and optimal configurations). In *Proceedings of the International Conference on Industrial Electronics, Control, and Instrumentation (IECON)*, (pp. 588–595). Bellingham, WA, USA: SPIE.

Furusawa, C., & Kaneko, K. (1998). Emergence of multicellular organisms with dynamic differentiation and spatial pattern. *Artificial Life*, *4*(1), 79–93.

Furusawa, C., & Kaneko, K. (2002). Origin of multicellular organisms as an inevitable consequence of dynamical systems. *The Anatomical Record*, *268*(3), 327–342.

Gazi, V., & Fidan, B. (2006). Coordination and control of multi-agent dynamic systems: models and approaches. In *Proceedings of the Second International Workshop on Swarm Robotics at SAB 2006*, vol. 4433 of *Lecture Notes in Computer Science*, (pp. 71–102). Rome, Italy: Springer.

Gazit, E. (2007). *Plenty of room for biology at the bottom: an introduction to bionanotechnology*. Imperial College Press.

Goldberg, D., Korb, B., & Deb, K. (1989). Messy genetic algorithms: motivation, analysis, and first results. *Complex systems*, *3*(5), 493–530.

Gómez-López, M., & Stoddart, J. (2002). Molecular and supramolecular nanomachines. In H. Nalwa (Ed.) *Nanostructured Materials and Nanotechnology*. Elsevier.

Goresky, M., & Klapper, A. (2009). Algebraic shift register sequences. *Unpublished manuscript*. URL `http://www.cs.uky.edu/~klapper/algebraic.html`

Görnerup, O., & Crutchfield, J. (2008). Hierarchical self-organization in the finitary process soup. *Artificial Life*, *14*(3), 245–254.

Goss, P., & Peccoud, J. (1998). Quantitative modeling of stochastic systems in molecular biology by using stochastic Petri nets. *Proceedings of the National Academy of Sciences of the USA (PNAS)*, *95*(12), 6750.

Gray, F. (1953). Pulse code communication. *United States Patent Office*, (Patent No. 2632058).

Groß, D., & Lenaerts, T. (2003). Towards a definition of dynamical hierarchies. In *Workshop Proceedings of Artificial Life VIII*, (pp. 45–55). Sydney, Australia: University of New South Wales Press.

Groß, D., & McMullin, B. (2001). Is it the right ansatz? *Artificial Life*, *7*(4), 355–365.

Grushin, A., & Reggia, J. (2008). Automated design of distributed control rules for the self-assembly of prespecified artificial structures. *Robotics and Autonomous Systems*, *56*(4), 334–359.

Haken, H. (1987). Self-organization and information. *Physica Scripta*, *35*, 247–254.

Hamad-Schifferli, K., Schwartz, J., Santos, A., Zhang, S., & Jacobson, J. (2002). Remote electronic control of DNA hybridization through inductive coupling to an attached metal nanocrystal antenna. *Nature*, *415*(6868), 152–155.

Hamlin, G., & Sanderson, A. (1997). TETROBOT: a modular approach to parallel robotics. *IEEE Robotics and Automation Magazine*, *4*(1), 42–51.

Harding, S., & Banzhaf, W. (2008). Artificial development. In R. Wuertz (Ed.) *Organic Computing*, Complex Systems Series, (pp. 201–220). Springer.

Harvey, I. (1996). The microbial genetic algorithm. *Unpublished manuscript*. URL `http://www.cogs.susx.ac.uk/users/inmanh/Microbial.pdf`

Harvey, I. (2001). Artificial evolution: A continuing SAGA. In T. Gomi (Ed.) *Proceedings of the 8th International Symposium on Evolutionary Robotics (ER2001)*, vol. 2217 of *Lecture Notes in Computer Science*, (pp. 94–109). Tokyo, Japan: Springer.

Harvey, I. (2009). The microbial GA. In *Proceedings of the European Conference on Artificial Life (ECAL)*, Lecture Notes in Computer Science, (p. to appear). Budapest, Hungary: Springer.

Harvey, I., Husbands, P., Cliff, D., Thompson, A., & Jakobi, N. (1997). Evolutionary robotics: the Sussex approach. *Robotics and Autonomous Systems*, *20*(2), 205–224.

Harvey, I., & Thompson, A. (1996). Through the labrynth evolution finds a way: a silicon ridge. In T. Higuchi, M. Iwata, & W. Liu (Eds.) *Proceedings of the First International Conference on Evolvable Systems: From Biology to Hardware (ICES)*, (pp. 406–422). Tsukuba, Japan: Springer-Verlag.

Hawking, S. (1988). *A Brief History of Time*. Bantam Books.

Helvik, T. (2005). *Dynamical systems of interacting units: information transport and higher order structures*. Ph.D. thesis, Norwegian University of Science and Technology.

Hlavacek, W., Faeder, J., Blinov, M., Posner, R., Hucka, M., & Fontana, W. (2006). Rules for modeling signal-transduction systems. *Science Signal Transduction Knowledge Environment (STKE)*, *334*(334), 1–18.

Hoekstra, A., Lorenz, E., Falcone, J., & Chopard, B. (2007). Towards a complex automata framework for multi-scale modeling: formalism and the scale separation map. In *Proceedings of the Seventh International Conference on Computational Science (ICCS)*, vol. 4487 of *Lecture Notes in Computer Science*, (pp. 922–930). Beijing, China: Springer.

Hofstadter, D. (1979). *Gödel, Escher, Bach: an Eternal Golden Braid*. Basic Books.

Holland, J. (1975). *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press.

Holland, J. (1992). *Adaptation in Natural and Artificial Systems*. Cambridge, MA: MIT Press.

Hutton, T. (2002). Evolvable self-replicating molecules in an artificial chemistry. *Artificial Life*, *8*(4), 341–356.

Hutton, T. (2007). Evolvable self-reproducing cells in a two-dimensional artificial chemistry. *Artificial Life*, *13*(1), 11–30.

Ikegami, T. (1999). Evolvability of machines and tapes. *Artificial Life and Robotics*, *3*(4), 242–245.

Ikegami, T., & Hashimoto, T. (1997). Replication and diversity in machine-tape coevolutionary systems. In *Artificial Life V*, (pp. 426–433). Nara, Japan: MIT Press.

Iocchi, L., Nardi, D., & Salerno, M. (2001). Reactivity and deliberation: a survey on multi-robot systems. *Balancing Reactivity and Social Deliberation in Multi-Agent Systems*, *2103*, 9–32.

Ishiguro, A., Shimizu, M., & Kawakatsu, T. (2006). A modular robot that exhibits ameobic locomotion. *Robotics and Autonomous Systems*, *54*(8), 641–650.

Jacobi, M. (2005). Hierarchical organization in smooth dynamical systems. *Artificial Life*, *11*(4), 493–512.

Jakobi, N. (1995). Harnessing morphogenesis. Cognitive Science Research Papers 423, University of Sussex, Falmer, UK.
URL `ftp://ftp.informatics.sussex.ac.uk/pub/reports/csrp/csrp423.ps.Z`

Jensen, K., & Rozenberg, G. (1991). *High-level Petri Nets*. Berlin: Springer-Verlag.

Jones, N., Landweber, L., & Lien, Y. (1977). Complexity of some problems in Petri nets. *Theoretical Computer Science*, *4*(3), 277 – 299.

Jørgensen, M., Østergaard, E., & Lund, H. (2004). Modular ATRON: modules for a self-reconfigurable robot. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, vol. 2, (pp. 2068–2073). Sendai, Japan: IEEE Press.

Kauffman, S. (1993). *The Origins of Order: Self-Organization and Selection in Evolution*. New York: Oxford University Press.

Keller, B., & Lutz, R. (2005). Evolutionary induction of stochastic context free grammars. *Pattern Recognition*, *38*(9), 1393–1406.

Kennedy, W., & Gentle, J. (1980). *Statistical Computing*. Statistics: Textbooks and Monographs. Marcel Dekker, Inc.

Kindler, E., & Weber, M. (2001). The Petri net kernel - an infrastructure for building Petri net tools. *International Journal on Software Tools for Technology Transfer (STTT)*, *3*(4), 486–497.

Kirby, B., Aksak, B., Hoburg, J., Mowry, T., & Pillai, P. (2007). A modular robotic system using magnetic force effectors. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS)*, (pp. 2787 – 2793). San Diego, CA, USA: IEEE Press.

Kitagawa, J., & Iba, H. (2003). Identifying metabolic pathways and gene regulation networks with evolutionary algorithms. In *Evolutionary Computation in Bioinformatics*, (pp. 255–274). Elsevier Science.

Klavins, E. (2006). Self-assembly from the point of view of its pieces. In *American Control Conference (ACC)*, (pp. 22–28). Minneapolis, MN, USA: IEEE Press.

Klavins, E., Burden, S., & Napp, N. (2006a). Optimal rules for programmed stochastic self-assembly. In G. Sukhatme, S. Schaal, W. Burgard, & D. Fox (Eds.) *Robotics: Science and Systems II (RSS)*, (pp. 9–16). Philadelphia, PA, USA: MIT Press.

Klavins, E., Ghrist, R., & Lipsky, D. (2006b). A grammatical approach to self-organizing robotic systems. *IEEE Transactions on Automatic Control*, *51*(6), 949–962.

Klein, M., & Shinoda, W. (2008). Large-scale molecular dynamics simulations of self-assembling systems. *Science*, *321*(5890), 798–800.

Kondacs, A. (2003). Biologically-inspired self-assembly of two-dimensional shapes using global-to-local compilation. In *International Joint Conference on Artificial Intelligence (IJCAI)*, (pp. 633–638). Acapulco, Mexico: Morgan Kauffman.

Kornienko, S., Kornienko, O., Nagarathinam, A., & Levi, P. (2007). From real robot swarm to evolutionary multi-robot organism. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*, (pp. 1483–1490). Singapore: IEEE Press.

Kotay, K., & Rus, D. (2000). Scalable parallel algorithm for configuration planning for self-reconfiguring robots. In G. McKee, & P. Schenker (Eds.) *Proceedings of Sensor Fusion and Decentralized Control in Robotic Systems III*, vol. 4196, (pp. 377–387). Boston, MA, USA: SPIE Press.

Kotay, K., Rus, D., Vona, M., & McGray, C. (1998). The self-reconfiguring robotic molecule. In *Proceedings of the IEEE Conference on Robotics and Automation (ICRA)*, vol. 4, (pp. 424–431). Leuven, Belgium: IEEE Press.

Koza, J. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.

Koza, J., Keane, M., Streeter, M., Mydlowec, W., Yu, J., & Lanza, G. (2005). *Genetic Programming IV*. Springer.

Kozen, D. (1997). *Automata and Computability*. New York: Springer-Verlag.

Krishnan, M., Tolley, M., Lipson, H., & Erickson, D. (2007). Directed hierarchical self assembly - active fluid mechanics at the micro and nanoscales. In *Proceedings of the ASME International Mechanical Engineering Congress and Exposition (IMECE)*. Seattle, WA, USA.

Kumar, S., & Bentley, P. (2003). Computational embryology: past, present and future. In *Advances in Evolutionary Computing: Theory and Applications*, Natural Computing Series, (pp. 461–477). Springer.

Kwong, H., & Jacob, C. (2003). Evolutionary exploration of dynamic swarm behaviour. In *IEEE Conference on Evolutionary Computation (CEC)*, (pp. 367–374). Canberra, Australia: IEEE Press.

Langton, C. (1984). Self-Reproduction in cellular automata. *Physica D: Nonlinear Phenomena*, *10*(1-2), 135–144.

Langton, C. (Ed.) (1989). *Artificial Life*. Redwood City: Addison-Wesley Publishing.

Lenaerts, T., Chu, D., & Watson, R. (2005). Dynamical hierarchies. *Artificial Life*, *11*(4), 403–405.

Lenski, R., Ofria, C., Pennock, R., & Adami, C. (2003). The evolutionary origin of complex features. *Nature*, *423*, 139–144.

Lin, C., & Marinescu, D. (1988). Stochastic high-level Petri nets and applications. *IEEE Transactions on Computers*, *37*(7), 815–825.

Livstone, M., Weiss, R., & Landweber, L. (2006). Automated design and programming of a microfluidic DNA computer. *Natural Computing*, *5*(1), 1–13.

Lucas, S., & Reynolds, T. (2007). Learning finite-state transducers: evolution versus heuristic state merging. *IEEE Transactions on Evolutionary Computation*, *11*(3), 308–325.

Luke, S., Cioffi-Revilla, C., Panait, L., & Sullivan, K. (2004). MASON: a new multi-agent simulation toolkit. In *Proceedings of the 2004 SwarmFest Workshop*, vol. 8. Ann Arbor, MI, USA.

Magnus, P. D. (2009). *forall x - An Introduction to Formal Logic*. University at Albany, State University of New York.
URL `http://www.fecundity.com/logic`

Mamei, M., Vasirani, F., & Zambonelli, F. (2004). Experiments of morphogenesis in swarms of simple robots. *Applied Artificial Intelligence*, *18*(10), 903–919.

Mataric, M., & Cliff, D. (1996). Challenges in evolving controllers for physical robots. *Robotics and Autonomous Systems*, *19*(1), 67–83.

Matsumaru, N., Centler, F., di Fenizio, P., & Dittrich, P. (2005). Chemical organization theory as a theoretical base for chemical computing. In C. Teusher, & A. Adamatzky (Eds.) *Proceedings of the 2005 Workshop on Unconventional Computing: From Cellular Automata to Wetware*, (pp. 75–88). Sevilla, Spain: Luniver Press.

Maturana, H., & Varela, F. (1980). *Autopoesis and Cognition*. Holland: Reidel Publishing.

Mayer, B., & Rasmussen, S. (1998). Self-reproduction of dynamical hierarchies in chemical systems. In C. Adami, R. Belew, H. Kitano, & C. Taylor (Eds.) *Artificial Life VI*, (pp. 123–129). Los Angeles, CA, USA: MIT Press.

Maynard Smith, J., & Szathmáry, E. (2004). *The major transitions in evolution*. Oxford University Press.

McCaskill, J. S. (1988). Polymer chemistry on tape: a computational model for emergent genetics. Internal report, Max Planck Institute for Biophysical Chemistry, Göttingen.

McGregor, S., & Fernando, C. (2005). Levels of description: a novel approach to dynamical hierarchies. *Artificial Life*, *11*(4), 459–472.

McMillan, K. (1995). A technique of state space search based on unfolding. *Formal Methods in System Design*, *6*(1), 45–65.

McNew, J., & Klavins, E. (2008). Non-deterministic reconfiguration of tree formations. In *American Control Conference (ACC)*, (pp. 690–697). Seattle, WA, USA: IEEE Press.

Melamed, B. (1998). *Design and implementation of an attribute manager for conditional and distributed graph transformation*. Master's thesis, Technical University of Berlin.
URL `http://user.cs.tu-berlin.de/~gragra/agg/Diplomarbeiten/Melamed.ps.gz`

Merkle, R. (1992). Self-replicating systems and molecular manufacturing. *Journal of the British Interplanetary Society*, *45*, 407–413.

Merkle, R. (1997). Convergent assembly. *Nanotechnology*, *8*(1), 18–22.

Mermoud, G., Brugger, J., & Martinoli, A. (2009). Towards multi-level modeling of self-assembling intelligent micro-systems. In *Proceedings of The Eighth International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, (pp. 89–96). Budapest, Hungary: International Foundation for Autonomous Agents and Multiagent Systems.

Mesarović, M., & Macke, D. (1969). Scientific theory of hierarchical systems. In L. Whyte, A. Wilson, & D. Wilson (Eds.) *Hierarchical Structures*, (pp. 29–50). Elsevier.

Miller, J., & Banzhaf, W. (2003). Evolving the program for a cell: from French flags to Boolean circuits. In S. Kumar, & P. Bentley (Eds.) *On Growth, Form and Computers*, (pp. 278–301). Elsevier.

Miller, J., Thomson, P., & Fogarty, T. (1997). Designing electronic circuits using evolutionary algorithms. arithmetic circuits: a case study. In K. Miettinen, P. Neittaanmäki, & M. Mäkelä (Eds.) *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science*, Computational Methods in Applied Sciences, (pp. 105–131). Wiley.

Milner, R. (2009). *The Space and Motion of Communicating Agents*. Cambridge University Press.

Milutinovic, D., & Lima, P. (2002). Petri net models of robotic tasks. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, vol. 4, (pp. 4059–4064). Washington, DC, USA: IEEE Press.

Mitchell, M. (1998). *An Introduction to Genetic Algorithms*. MIT Press.

Molloy, M. (1982). Performance analysis using stochastic Petri nets. *IEEE Transactions on Computers*, *C-31*(9), 913–917.

Moore, J. H., & Hahn, L. W. (2004a). Evaluation of a discrete dynamic systems approach for modeling the hierarchical relationship between genes, biochemistry, and disease susceptibility. *Discrete and Continuous Dynamical Systems*, *4*(1), 275–288.

Moore, J. H., & Hahn, L. W. (2004b). Systems biology modeling in human genetics using Petri nets and grammatical evolution. In *Genetic and Evolutionary Computation (GECCO)*, (pp. 392–401).

Murata, S., Kamimura, A., Kurokawa, H., Yoshida, E., Tomita, K., & Kokaji, S. (2004). Self-reconfigurable robots: platforms for emerging functionality. *Embodied Artificial Intelligence*, *3139*, 312–330.

Murata, S., Kurokawa, H., & Kokaji, S. (1994). Self-Assembling machine. In *Proceedings of the IEEE Conference in Robotics and Automation (ICRA)*, (pp. 441–448). San Diego, CA, USA: IEEE Press.

Nagpal, R., Kondacs, A., & Chang, C. (2003). Programming methodology for biologically-inspired self-assembling systems. In *AAAI Spring Symposium on Computational Synthesis: From Basic Building Blocks to High Level Functionality*. Stanford University, CA, USA: AAAI Press.

Nehaniv, C., & Rhodes, J. (2000). The evolution and understanding of hierarchical complexity in biology from an algebraic perspective. *Artificial Life*, *6*(1), 45–67.

Nelson, A., Barlow, G., & Doitsidis, L. (2009). Fitness functions in evolutionary robotics: A survey and analysis. *Robotics and Autonomous Systems*, *57*(4), 345–370.

Nolfi, S., & Parisi, D. (2002). Evolution of artificial neural networks. In M. Arbib (Ed.) *Handbook of Brain Theory and Neural Networks*, (pp. 418–421). MIT Press.

Nummela, J., & Julstrom, B. (2005). Evolving Petri nets to represent metabolic pathways. In H. Beyer, & U. O'Reilly (Eds.) *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation (GECCO)*, (pp. 2133–2139). Washington, DC, USA: ACM.

Ofria, C., & Adami, C. (1999). Evolution of genetic organization in digital organisms. In L. Landweber, & E. Winfree (Eds.) *Evolution as Computation: DIMACS Workshop*, Natural Computing Series, (pp. 296–313). Princeton, NJ: Springer-Verlag.

Ono, N., & Ikegami, T. (2001). Artificial chemistry: computational studies on the emergence of self-replicating units. In J. Kelemen, & P. Sosík (Eds.) *Proceedings of the European Conference on Artificial Life (ECAL)*, vol. 2159 of *Lecture Notes in Computer Science*, (pp. 186–195). Prague, Czech Republic: Springer.

Østergaard, E., & Lund, H. (2003). Evolving control for modular robotic units. In *Proceedings of the IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA)*, (pp. 886–893). Kobe, Japan: IEEE Press.

Ota, J. (2006). Multi-agent robot systems as distributed autonomous systems. *Advanced Engineering Informatics*, *20*(1), 59–70.

Palamara, P. F., Ziparo, V. A., Iocchi, L., Nardi, D., & Lima, P. (2009). Teamwork design based on Petri net plans. In L. Iocchi, H. Matsubara, A. Weitzenfeld, & C. Zhou (Eds.) *RoboCup 2008: Robot Soccer World Cup XII*, Lecture Notes in Artificial Intelligence, (pp. 211–222). Suzhou, China: Springer.

Parker, L. (2000). Current state of the art in distributed autonomous mobile robotics. *Distributed Autonomous Robotic Systems*, *4*, 3–12.

Pask, G. (1958a). The growth process inside the cybernetic machine. In *Second International Conference on Cybernetics*, (pp. 765–794). Namur, Belgium.

Pask, G. (1958b). Physical analogues to the growth of a concept. In *Mechanization of Thought Processes, Symposium 10*, (pp. 765–794). National Physical Laboratory, London: H. M. S. O.

Peleg, M., Rubin, D., & Altman, R. (2005). Using Petri net tools to study properties and dynamics of biological systems. *Journal of the American Medical Informatics Association*, *12*(2), 181–199.

Penrose, L. (1959). Self-reproducing machines. *Scientific American*, *206*(6), 105–114.

Peterson, J. (1981). *Petri Net Theory and the Modeling of Systems*. Upper Saddle River, NJ, USA: Prentice Hall PTR.

Petri, C. (1962). *Kommunikation mit Automaten*. Ph.D. thesis, Technische Hochschule Darmstadt, Darmstadt, Germany.

Petri, C. (1996). Nets, time, and space. *Theoretical Computer Science*, *153*, 3–48.

Petri, C. (2008). On the physical basics of information flow - results obtained in cooperation with Konrad Zuse. In *Petri Nets*, vol. 5062 of *Lecture Notes in Computer Science*. Xi'an, China: Springer.

Petri, C., & Reisig, W. (2008). Petri net. *Scholarpedia*, *3*(4), 6477.
URL `http://www.scholarpedia.org/article/Petri_net`

Petri, C., & Smith, E. (1987). "Forgotten topics" of net theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, vol. 255 of *Lecture Notes in Computer Science*, (pp. 499–514). Springer.

Pfeifer, R., & Iida, F. (2004). Embodied artificial intelligence: Trends and challenges. In *Embodied Artificial Intelligence*, vol. 3139 of *Lecture Notes in Artificial Intelligence*, (pp. 1–26).

Prencipe, G., & Santoro, N. (2006). Distributed algorithms for autonomous mobile robots. In G. Navarro, N. Bertossi, & Y. Kohayakawa (Eds.) *Proceedings of 5th IFIP International Conference on Theoretical Computer Science (TCS)*, vol. 209 of *IFIP*, (pp. 47–62). Santiago, Chile: Springer.

Păun, G., & Rozenberg, G. (2002). A guide to membrane computing. *Theoretical Computer Science*, *287*(1), 73–100.

Pugh, J., & Martinoli, A. (2006). Multi-robot learning with particle swarm optimization. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, (pp. 448–453). Hakodate, Japan: The International Foundation for Autonomous Agents and Multiagent Systems.

Rasmussen, S., Baas, N., Mayer, B., & Nilsson, M. (2001a). Defense of the ansatz for dynamical hierarchies. *Artificial Life*, *7*(4), 367–373.

Rasmussen, S., Baas, N., Mayer, B., Nilsson, M., & Olesen, M. (2001b). Ansatz for dynamical hierarchies. *Artificial Life*, *7*(4), 329–353.

Ray, T. (1992). Evolution, ecology and optimization of digital organisms. Working Paper 92-08-042, Santa Fe Institute.
URL `http://life.ou.edu/pubs/tierra/`

Ray, T. (1997). Evolving complexity. *Artificial Life and Robotics*, *1*(1), 21–26.

Regev, A., & Shapiro, E. (2002). Cells as computation. *Nature*, *419*, 343.

Regev, A., Silverman, W., & Shapiro, E. (2001). Representation and simulation of biochemical processes using the π-calculus process algebra. In *Pacific Symposium on Biocomputing*, vol. 6, (pp. 459–470). Mauna Lani, HI, USA: World Scientific Press.

Reid, D. (1998). Constructing Petri net models using genetic search. *Mathematical and Computer Modeling*, *27*(8), 85 – 103.

Reisig, W. (1992). *A Primer in Petri Net Design*. Berlin: Springer-Verlag.

Reisig, W. (2009). The universal net composition operator. Research report, Humboldt University of Berlin.
URL `http://www2.informatik.hu-berlin.de/top/publikationen/de/Author/REISIG-W.php`

Roggen, D., & Federici, D. (2004). Multi-cellular development: is there scalability and robustness to gain? In *Parallel Problem Solving from Nature (PPSN VIII)*, vol. 3242 of *Lecture Notes in Computer Science*, (pp. 391–400). Birmingham, UK: Springer.

Ronse, C. (1982). *Feedback Shift Registers*. Berlin: Springer-Verlag.

Rothemund, P. (2006). Folding DNA to create nanoscale shapes and patterns. *Nature*, *440*, 297–302.

Rothemund, P., Papadakis, N., & Winfree, E. (2004). Algorithmic self-assembly of DNA sierpinski triangles. *PLoS Biology*, *2*(12), e424.

Rowe, J., Vose, M., & Wright, A. (2005). State aggregation and population dynamics in linear systems. *Artificial Life*, *11*(4), 473–492.

Rudolf, M. (1997). *Konzeption und implementierung eines interpreters für atrributierte graphtransformation*. Ph.D. thesis, Technical University of Berlin.

Rus, D., & Vona, M. (1999). Self-reconfiguration planning with compressible unit modules. In *Proceedings of the 1999 IEEE International Conference on Robotics and Automation (ICRA)*, vol. 4, (pp. 2513–2520). Detroit, MI, USA: IEEE Press.

Saitou, K. (1999). Conformational switching in self-assembling mechanical systems. *IEEE Transactions on Robotics and Automation*, *15*(3), 510–520.

Saitou, K., Malpathak, S., & Qvam, H. (2002). Robust design of flexible manufacturing systems using colored Petri net and genetic algorithm. *Journal of Intelligent Manufacturing*, *13*(5), 339–351.

Saksida, L. M., Raymond, S. M., & Touretzky, D. S. (1997). Shaping robot behavior using principles from instrumental conditioning. *Robotics and Autonomous Systems*, *22*, 231–250.

Salzberg, C. (2006). Complexity scaling of a minimal functional chemistry. In *Artificial Life X*, (pp. 165–171). Bloomington, IN, USA: MIT Press.

Salzberg, C. (2007). A graph-based reflexive artificial chemistry. *BioSystems*, *87*(1), 1–12.

Sayama, H. (2009). Swarm chemistry. *Artificial Life*, *15*(1), 105–114.

Seelig, G., Soloveichik, D., Zhang, D., & Winfree, E. (2006). Enzyme-free nucleic acid logic circuits. *Science*, *314*(5805), 1585–1588.

Sheng, W., & Yang, Q. (2005). Peer-to-peer multi-robot coordination algorithms: Petri net based analysis and design. In *Proceedings of the IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM)*, (pp. 1407–1413). Monterey, CA: IEEE Press.

Shirayama, M., Koshino, M., Hatakeyama, T., & Kimura, H. (2004). Artificial life simulation of self-assembly in bacteriophage by movable finite automata. *BioSystems*, *77*(1-3), 151–161.

Sims, K. (1994). Evolving virtual creatures. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, (pp. 15–22). Orlando, FL, USA: ACM.

Sipper, M. (1998). Fifty years of research on self-replication: an overview. *Artificial Life*, *4*(3), 237–257.

Smith, A., Turney, P., & Ewaschuk, R. (2003). Self-replicating machines in continuous space with virtual physics. *Artificial Life*, *9*(1), 21–40.

Smith, R., & Cribbs, H. I. (1994). Is a learning classifier system a type of neural network? *Evolutionary Computation*, *1*(2), 19–36.

Smullyan, R. (1995). *First-order logic*. Dover Publications.

Standish, R. (2003). Open-ended artificial evolution. *International Journal of Computational Intelligence and Applications*, *3*, 167–175.

Stanley, K., & Miikkulainen, R. (2003). A taxonomy for artificial embryogeny. *Artificial Life*, *9*(2), 93–130.

Støy, K., & Nagpal, R. (2004). Self-repair through scale independent self-reconfiguration. In *Proceedings of the 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, vol. 2, (pp. 2062–2067). Sendai, Japan: IEEE Press.

Støy, K., Shen, W., & Will, P. (2003). A simple approach to the control of locomotion in self-reconfigurable robots. *Robotics and Autonomous Systems*, *44*(3-4), 191–199.

Studer, G., & Harvey, I. (2007). A distributed formation algorithm to organize agents with no coordinate agreement. vol. 4648 of *Lecture Notes in Computer Science*, (pp. 515–524). Lisbon, Portugal: Springer.

Studer, G., & Harvey, I. (2008). A minimal approach to modular assembly. In *Artificial Life XI, as presented abstract*. Winchester, UK.

Studer, G., & Lipson, H. (2006). Spontaneous emergence of self-replicating structures in molecube automata. In *Artificial Life X*, (pp. 227–233). Bloomington, IN, USA: MIT Press.

Suh, J., Homans, S., & Yim, M. (2002). Telecubes: Mechanical design of a module for self-reconfigurable robotics. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, vol. 4, (pp. 4095–4101). Washington, DC, USA: IEEE Press.

Suzuki, I., & Yamashita, M. (1996). Distributed anonymous mobile robots - formation and agreement problems. In *Proceedings of the Third Colloqium on Structural Information and Communication Complexity (SIROCCO)*, (pp. 313–330). Siena, Italy: Carleton Scientific.

Tan, K., Wang, L., Lee, T., & Vadakkepat, P. (2004). Evolvable hardware in evolutionary robotics. *Autonomous Robots*, *16*(1), 5–21.

Theraulaz, G., & Bonabeau, E. (1995). Coordination in distributed building. *Science*, *269*(5224), 686–688.

Thompson, R., & Goel, N. (1988). Movable finite automata (MFA) models for biological systems I: bacteriophage assembly and operation. *Journal of Theoretical Biology*, *131*(3), 351–385.

Thorsley, D., & Klavins, E. (2008). Model reduction of stochastic processes using Wasserstein pseudometrics. In *American Control Conference (ACC)*, (pp. 1374–1381). Seattle, WA, USA: IEEE Press.

Tohme, H., Nakamura, M., Hachiman, E., & Onaga, K. (1999). Evolutionary Petri net approach to periodic job-shop-scheduling. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, vol. 4, (pp. 441–446). Tokyo, Japan: IEEE Press.

Tolley, M., Krishnan, M., Erickson, D., & Lipson, H. (2008). Dynamically programmable fluidic assembly. *Applied Physics Letters*, *93*(25), 254105.

Trianni, V., Groß, R., Labella, T., Şahin, E., & Dorigo, M. (2003). Evolving aggregation behaviors in a swarm of robots. In *Proceedings of the European Conference on Artificial Life (ECAL)*, vol. 2801 of *Lecture Notes in Computer Science*, (pp. 865–874). Dortmund, Germany: Springer.

Trianni, V., Nolfi, S., & Dorigo, M. (2004). Hole avoidance: experiments in coordinated motion on rough terrain. In F. Groen, N. Amato, A. Bonarini, E. Yoshida, & B. Krose (Eds.) *Intelligent Autonomous Systems (IAS) 8*, (pp. 29–36). Amsterdam, The Netherlands: IOS Press.

Tuci, E., Groß, R., Trianni, V., Mondada, F., Bonani, M., & Dorigo, M. (2005). Cooperation through self-assembly in multi-robot systems. Tech. Rep. TR/IRIDIA/2005-3, Universite Libre de Bruxelles.
URL `http://www.swarm-bots.org/`

Ulam, S. (1950). Random processes and transformations. In *Proceedings of the International Congress of Mathematicians*, vol. 2, (pp. 264–275). Cambridge, MA.

Ünsal, C., & Khosla, P. K. (2000). Mechatronic design of a modular self-reconfiguring robotic system. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, (pp. 1742–1747). San Francisco, CA: IEEE Press.

Valmari, A. (1998). The state explosion problem. In *Lectures on Petri Nets I: Basic Models*, vol. 1491 of *Lecture Notes in Computer Science*, (pp. 429–528). Springer-Verlag.

Varela, F., Thompson, E., & Rosch, E. (1992). *The Embodied Mind: Cognitive Science and Human Experience*. The MIT Press.

Von Neumann, J. (1963). The general and logical theory of automata. In *Collected Works of John von Neumann*, vol. 5, (pp. 288–328). Pergamon Press.

Von Neumann, J. (1966). *Theory of Self-Reproducing Automata*. Champaign, IL: University of Illinois Press.

Wang, F., & Saridis, G. (1993). Task translation and integration specification in intelligent machines. *IEEE Transactions on Robotics and Automation*, *9*(3), 257–271.

Weinberger, E. (1990). Correlated and uncorrelated fitness landscapes and how to tell the difference. *Biological Cybernetics*, *63*(5), 325–336.

Werfel, J., Bar-Yam, Y., Rus, D., & Nagpal, R. (2006). Distributed construction by mobile robots with enhanced building blocks. In *Proceedings of 2006 IEEE International Conference on Robotics and Automation, Orlando, USA*.

White, P., & Yim, M. (2007). Scalable modular self-reconfigurable robots using external actuation. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, (pp. 2773–2778). San Diego, CA, USA: IEEE Press.

White, P., Zykov, V., Bongard, J., & Lipson, H. (2005). Three dimensional stochastic reconfiguration of modular robots. In S. Thrun, G. Sukhatme, & S. Schaal (Eds.) *Robotics: Science and Systems (RSS) I*, (pp. 161–168). Cambridge, MA, USA: MIT Press.

Whitesides, G. (2002). Self-Assembly at all scales. *Science*, *295*(5564), 2418–2421.

Whitesides, G., Mathias, J., & Seto, C. (1991). Molecular self-assembly and nanochemistry: a chemical strategy for the synthesis of nanostructures. *Science*, *254*(5036), 1312–1319.

Winfree, E. (1996). On the computational power of DNA annealing and ligation. In L. Landweber, & E. Baum (Eds.) *DNA Based Computers II*, vol. 27 of *DIMACS: Discrete Math and Theoretical Computer Science*, (pp. 199–221). Princeton, NJ, USA: American Mathematics Society.

Wolpert, D., & Macready, W. (1997). No free lunch theorems for search. *IEEE Transactions on Evolutionary Computation*, *1*(1), 67–82.

Wolpert, L., & Dover, G. (1981). Positional information and pattern formation. *Biological Sciences*, *295*, 441–450.

Wu, G., Jonoska, N., & Seeman, N. (2009). Construction of a DNA nano-object directly demonstrates computation. *BioSystems*, *98*(2), 80–84.

Yang, J., Monine, M., Faeder, J., & Hlavacek, W. (2008). Kinetic Monte Carlo method for rule-based modeling of biochemical networks. *Physical Review E*, *78*(3), 31910.

Yao, X. (1999). Evolving artificial neural networks. *Proceedings of the IEEE*, *87*(9), 1423–1447.

Yao, X., & Higuchi, T. (1997). Promises and challenges of evolvable hardware. In *Evolvable Systems: From Biology to Hardware (Proceedings of the International Conference on Evolvable Systems) (ICES)*, vol. 5216 of *Lecture Notes in Computer Science*, (pp. 55–78). Prague, Czech Republic: Springer.

Yim, M., Duff, D., & Roufas, K. (2000). PolyBot: a modular reconfigurable robot. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, (pp. 514–521). San Francisco, CA: IEEE Press.

Yim, M., Shen, W., Salemi, B., Rus, D., Moll, M., Lipson, H., Klavins, E., & Chirikjian, G. (2007a). Modular self-reconfigurable robot systems. *IEEE Robotics & Automation Magazine*, *14*(1), 43–52.

Yim, M., Shirmohammadi, B., Sastra, J., Park, M., Dugan, M., & Taylor, C. (2007b). Towards robotic self-reassembly after explosion. In *Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, (pp. 2767–2772). San Diego, CA, USA: IEEE Press.

Yim, M., White, P., Park, M., & Sastra, J. (2009). Modular self-reconfigurable robots. In *Encyclopedia of Complexity and Systems Science*, (pp. 5618–5631). Springer.

Yoshida, H., Furusawa, C., & Kaneko, K. (2005). Selection of initial conditions for recursive production of multicellular organisms. *Journal of Theoretical Biology*, *233*(4), 501–514.

Zuse, K. (1969). Rechnender raum. *Shriften zur Datenverarbeitung*, *1*, 74.
URL `ftp://ftp.idsia.ch/pub/juergen/zuserechnenderraum.pdf`

Zykov, V., Mytilinaios, E., Desnoyer, M., & Lipson, H. (2007). Evolved and designed self-reproducing modular robotics. *IEEE Transactions on Robotics*, *23*(2), 308–319.