An Architectural Style for Solving Computationally Intensive Problems on Large Networks

Yuriy Brun and Nenad Medvidovic
Computer Science Department
University of Southern California
Los Angeles, California 90089, USA
{ybrun, neno}@usc.edu

Abstract

Large networks, such as the Internet, pose an ideal medium for solving computationally intensive problems, such as NP-complete problems, yet no well-scaling architecture for computational Internet-sized systems exists. We propose a software architectural style for large networks, based on a formal mathematical study of crystal growth that will exhibit properties of (1) discreetness (nodes on the network cannot learn the algorithm or input of the computation), (2) fault-tolerance (malicious, faulty, and unstable nodes may not break the computation), and (3) scalability (communication among the nodes does not increase with network or problem size).

1 Introduction

The Internet's growth has created networks with great computing potential without a clear way to harness that potential to solve memory- and processor time-intensive problems. Networks, such as the Internet, have the potential to solve NP-complete problems (and other problems for which we do not know polynomial time solutions) quickly, but as their individual nodes may be unreliable or malicious, users may desire guarantees that their computations are correct and are kept confidential. Mechanisms for distributing computation over such large networks are likely to require a great deal of collaboration, while the large size of the network is likely to require that collaboration to scale well.

The architectural style presented in this paper is particularly applicable to problems that are computationally intensive and easily parallelizable. Computationally intensive problems are ones that a single computer is unlikely to solve quickly, while easily parallelizable problems are ones that inherently yield a large number of parallel threads. For example, all NP-complete problems have both of those properties [18]. Further, our work is applicable to users that desire discreetness and have access to large but unreliable networks. By discreetness, we mean that the user does not want others to find out the input or the algorithm. By large but unreliable network, we mean a network, such as the Internet, that is partially or entirely outside of the user's control, and perhaps even hostile.

We describe a sample scenario that is at the heart of the problems we are tackling. An espionage agency is attempting to break an RSA code sent by an enemy. The agency wishes to use a large network to factor the enemy's public key; however, it cannot allow anyone to know the key's factors or even whose key it is factoring. Since the agency has access to the Internet, an incredibly large network of computers, it should be feasible to factor nondeterministically, or through brute force. However, the problem is to do so discreetly, without the nodes on the network learning the problem or the input.

The above scenario will result in a complex distributed software system. It has been shown that such systems are most effectively approached from an architectural perspective (e.g., [14]). In particular, software architectural *styles* present generic design solutions that can be applied to problems with shared characteristics.

We propose to create a software architectural style that allows distributing problems over a large network in a fault-tolerant, discreet, and scalable manner. To that end, we will rely on the theoretical study of self-assembly and a formal model of crystal growth, called the tile assemble model [20]. This model is Turing universal, thus it can compute all the functions that a traditional computer program can. Systems in this model show remarkable fault-tolerance, self-regeneration, distribution of information, and scalability, and a software architecture that implements the rules of such systems should inherit these properties.

The architectural style we present in this paper can

be evaluated theoretically, using mathematical analysis of the architecture, and empirically, using a system, designed based on the tile style, that simulates a large network solving an NP-complete problem. In this paper, we present preliminary mathematical theoretical evaluation.

2 Related Work

We propose an architectural style, called the tile style, for solving NP-complete problems on large networks. The tile style is based on a model of self-assembly, thus this section describes the related work of two previously mostly independent areas: software architectures and theoretical study of self-assembly.

2.1 Software Architectures

Software architecture has been identified as an important part of building almost all large systems [14]. A poor underlying software architecture can be disastrous, while a good one helps to ensure the system's key properties, such as performance, reliability, portability, scalability, and interoperability.

Software architecture can be used to "force" a software system to conform to certain rules, thus resulting in some desired properties. For example, mandating that two components communicate via implicit invocation can result in systems that are more easily evolvable. However, it is also possible to provide desired system properties as an emergent behavior of the architecture without forcing restrictions on the system designer. For example, Mikic-Rakic et al. have argued that for a system to be self-healing, the system must be self-observant and alter its behavior in hostile environments [13]. However, in our proposed architectural style, the system exhibits properties of self-healing naturally, without observing or altering its behavior. Similarly, Devanbu et al. have argued that security, a crucial property of most modern software systems, may be implemented in the connectors mediating the interactions among the system's components [11]. Accordingly, our architectural style allows for security in the connectors; however, discreetness, one aspect of security, is an *emergent* property of the style.

While there are several definitions of architectural styles (e.g., [6, 10, 17]), we directly leverage Mikic-Rakic et al.'s [13] definition, in formulating the tile style. They have argued that an architectural style can be described along five dimensions: external structure, topology rules, behavior, interaction, and data flow. External structure describes the "outside view" of the components in the architectural style; topology rules describe the allowed paths of interaction between those components; behavior describes the components' internal function and state; interaction captures the collaboration between the components; and data flow specifies the structure of the data exchanged by the components. We will follow this scheme in defining our architectural style.

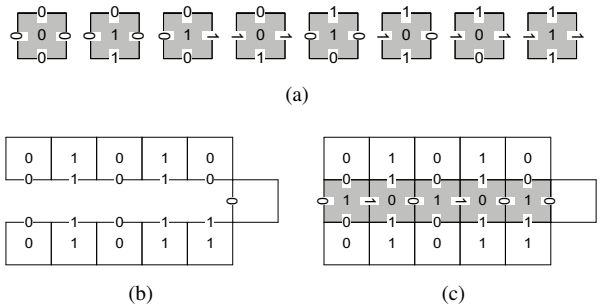

Figure 1. A sample tile system that adds numbers. (a) The system has eight computation tiles. (b) A seed configuration encodes the inputs, $10 = 1010_2$ and $11 = 1011_2$. (c) The gray computation tiles attach to the seed to form the output $21 = 10101_2$.

2.2 Self-Assembly

Adleman proposed solving NP-complete problems using DNA [1]. Winfree has generalized Adleman's ideas to use an exponential number of independent nodes, forming a formalized mathematical model of self-assembly, the tile assembly model [20]. The tile assembly model is a model of crystal growth that has been shown to be Turing universal [4, 19].

In the tile assembly model, individual components are square tiles with special labels on their four sides. Tiles can stick together under certain conditions when their abutting sides' labels match. It is possible to encode inputs using tiles and design sets of tiles that attach to those input tiles to compute functions [7]. Figure 1 shows a very simple example of adding the numbers 10 and 11, which we will explain in Section 3.1. Two physical implementations of the tile assembly model exist, one that computes the Sierpinski triangle [15] and another that counts in binary [5]; both use DNA complexes called double crossover complexes [12] to implement tiles.

Traditional computational measures, such as program size and time complexity, are related to tile systems: program size is related to the number of different tiles in a system and time consumption is related to the tile system's assembly time [2]. Adleman and others have studied the number of tiles and time steps it takes to assemble n-long linear polymers [2], shapes [3, 16], and simulate Turing machines [4, 19].

When Winfree developed the tile assembly model of crystal growth, he proposed a mechanism for controlling the

error rates, while preserving many of the aspects of DNA computation. He observed that errors occur in two situations: when incorrect tiles attach and when tiles attach in improper places. He [21] and Goel et al. [9] proposed solutions for dealing with such errors, reducing the probability of failure. Both techniques are linear-space transformations of the tiles of a system. That is, given a system with n types of tiles and a probability of making an error of p, they generate another system that has kn tile types, but its likelihood of failure is p^k.

Building on the above work, we have extended the notion of computing functions to the tile assembly model and studied systems that add and multiply [7]. We found that in the tile assembly model, adding and multiplying can be done using $\Theta(1)$ tiles (as few as 8 tiles for addition and as few as 28 tiles for multiplication), and that both computations can be carried out in time linear in the input size. Here, we present a tile system that solves a more complex problem, an NP-complete problem called SubsetSum, which can be used as a tool to solve all NP-complete problems.

3 The Tile Style Approach

This section describes the tile architectural style by (1) explaining how tile self-assembly can compute, (2) presenting a particular tile system that solves an NP-complete problem called SubsetSum, (3) explaining the design of a software architecture based on a tile system, and finally (4) presenting an example of using the system based on that software architecture to solve a specific instance of the SubsetSum problem.

In the tile architectural style, each physical node on the network will represent several tiles in a tile system. Because of the details of that representation, the systems built based on such architectures should inherit the properties of the tile systems: discreetness, fault-tolerance, and scalability.

3.1 Computing with Tiles

It may seem counterintuitive that simple tiles that only interact with each other locally can compute complex functions, but in fact, these tile systems are as powerful as every computer [19]. In this section, we show how it is possible for tile systems to compute.

3.1.1 Theoretical Underpinnings

Computing can be defined as manipulating data in a controlled manner to produce the results of a series of mathematical functions. While traditional computers use gates to manipulate data stored in electronic form, other computational models use other means of controlling the data. Self-assembly uses square tiles to encode information and compute. What follows is an intuitive description of the formal definitions of the tile assembly model, which will be necessary in mathematically proving properties of the systems built based on the tile architectural style. The reader may bypass this section if she is interested in the results and not the theoretical arguments.

Intuitively, the model has *tiles*, or squares, that stick or do not stick together based on various *binding domains* on their four sides. Each tile has a binding domain on its north, east, south, and west side. The four binding domains, elements of a finite alphabet Σ, define the type of the tile. The strength of the binding domains are defined by the *strength function* g. The placement of some tiles on a 2-D grid is called a *configuration*, and a tile may *attach* in empty positions on the grid if the total strength of all the binding domains on that tile that match its neighbors exceeds the current *temperature* (a natural number). Finally, a *tile system* $\mathbb{S}$ is a triple $\langle T, g, \tau \rangle$, where T is a finite set of tiles, g is a strength function, and $\tau \in \mathbb{N}$ is the temperature, where $\mathbb{N} = \mathbb{Z}_{\geq 0}$.

Starting from a *seed configuration* S, tiles may attach to form new configurations. If that process terminates, the resulting configuration is said to be *final*. At some times, it may be possible for more than one tile to attach at a given position, or there may be more than one position where a tile can attach. If for all sequences of tile attachments, all possible final configurations are identical, then $\mathbb{S}$ is said to produce a *unique* final configuration on S. The *assembly time* of the system is the minimal number of steps it takes to build a final configuration, assuming maximum parallelism.

Let f be a function $f : \mathbb{N}^n \rightarrow \mathbb{N}^m$. A tile system $\mathbb{S}$ is said to *deterministically compute* f if these exists a seed that encodes $\vec{a} \in \mathbb{N}^n$ and $\mathbb{S}$ produces a unique final configurations F that encodes $f(\vec{a})$. Similarly, a tile system $\mathbb{S}$ is said to *nondeterministically compute* f with identifier tile $r \in T$ iff for all $\vec{a} \in \mathbb{N}^n$, there exists a seed configuration S that encodes $\vec{a}$ and for all final configurations F that $\mathbb{S}$ produces on S, F contains r iff F encodes $f(\vec{a})$ and there exists at least one final configuration F that encodes $f(\vec{a})$. In other words, the *identifier* tile r only attaches to the successful nondeterministic executions that encode the solution.

We have given informal definitions to assist the reader in understanding the systems we discuss in this paper; we refer the reader to [7] for more formal versions of the definitions.

The tiles in Figure 1(a), with all binding domains having strength 1, and temperature 3, form a tile system that computes the function $f(a, b) = a + b$. That is, it is an adding system. Figure 1 shows an example of that tile system computing the sum of $10 = 1010_2$ and $11 = 1011_2$. The system has eight computational tile types (Figure 1(a)). The center state variable of each tile in Figure 1(c) represents one bit of the solution, and the west side represents the next carry bit. For example, the right-most tile in Figure 1(a) adds two 1

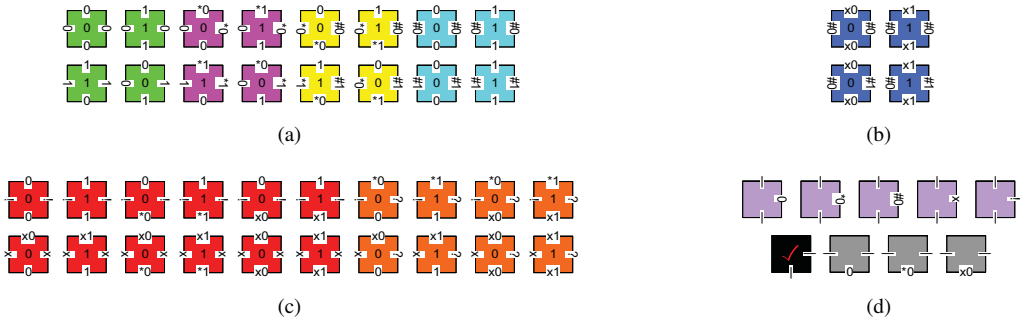

Figure 2. A tile system that solves SubsetSum consists of four smaller tile systems, designed to work together. (a) The tiles of a system that subtracts numbers, (b) the tiles of a system that simply copies information upward, (c) the tiles of a system that nondeterministically picks whether or not to subtract the next number, and (d) the tiles of a system that makes sure the subtractions completed correctly and the result is 0.

bits and has a 1 incoming carry bit, so it has a state value 1 and an outgoing carry bit 1. Starting from a seed configuration (Figure 1(b)), instances of the gray tile types can attach if their sides match the neighbors' sides. The final configuration (Figure 1(c)), encodes the answer $21 = 10101_2$ in the center row. We refer the reader to [7] for the full proof that this is an adding system.

3.1.2 SubsetSum Tile System

SubsetSum is a well-known NP-complete problem. The problem consists of determining whether the sum of a subset of numbers adds up to a given target number. The input to the problem is a a set of natural numbers and a natural target number, and the output is 1 if the sum of some subset of those numbers is equal to the target number, and 0 otherwise.

The nature of NP-complete problems is that if one can solve one such problem quickly, then one can solve all such problems quickly. For example, if one finds a polynomial time algorithm to solve SubsetSum, one can now solve the traveling salesman, 3-SAT, and all other NP problems in polynomial time. Thus, it is sufficient to design a system that uses a large distributed network to discreetly solve one NP-complete problem, e.g., SumbsetSum. We present a tile system that solves SubsetSum. Winfree has shown a way to translate an arbitrary computer program into a tile system [19]; however, that translation can be inefficient. The system we describe here for solving SubsetSum is the result of our engineering and design efforts to create a system that computes quickly and uses a small set of tile types, a process similar to writing a computer program.

The SubsetSum tile system is really a combination of four tile systems, designed to work together. Figures 2(a-d) show the tiles of systems that subtracts numbers, copies a number (subtract 0), nondeterministically picks whether or not to subtract the next number, and checks if all the subtractions completed correctly and if the final result is 0, respectively. The system nondeterministically picks a subset of the input numbers to subtract from the target number, and if the result equals 0, attaches a special ✓ tile.

Figure 3 shows a sample execution of the tile system that solves SubsetSum. The example asks the question whether or not the sum of some subset of the set $\{11, 25, 37, 39\}$ equals 75. Because $75 = 11+25+39$, one nondeterministic execution of the tile system finds the proper selection of numbers and attaches the special ✓ tile. If there were no subset of numbers whose sum equaled 75, no such tile could attach. We refer the reader to [8] for the full proof that this system solves the SubsetSum problem.

3.2 Tile Architectural Style

A tile style architecture is based on a tile system. The components of the architecture are instantiations of the tile types. While a system based on such an architecture will have a large number of components, there is a comparatively smaller number of different *types* of components (e.g., 8 types for adding and 49 types for solving SubsetSum). Nodes on the network represent these components, and components that are adjacent in an assembly (such as those in Figure 1(b)), can *recruit* other components to attach, by sampling nodes until they find one whose side labels, or interfaces, match. Note that many components (i.e. tiles) can run on a single physical node.

In addition to defining the tile types, a tile system also directs the architecture how to encode the input to the computation. The input consists of a seed, a small connected collection of tiles, such as the clear tiles along the right and bottom edges in Figure 3. The seed replicates on the network, as described in section 3.2.1, to create many copies.

Each component's externally visible *structure* is a state

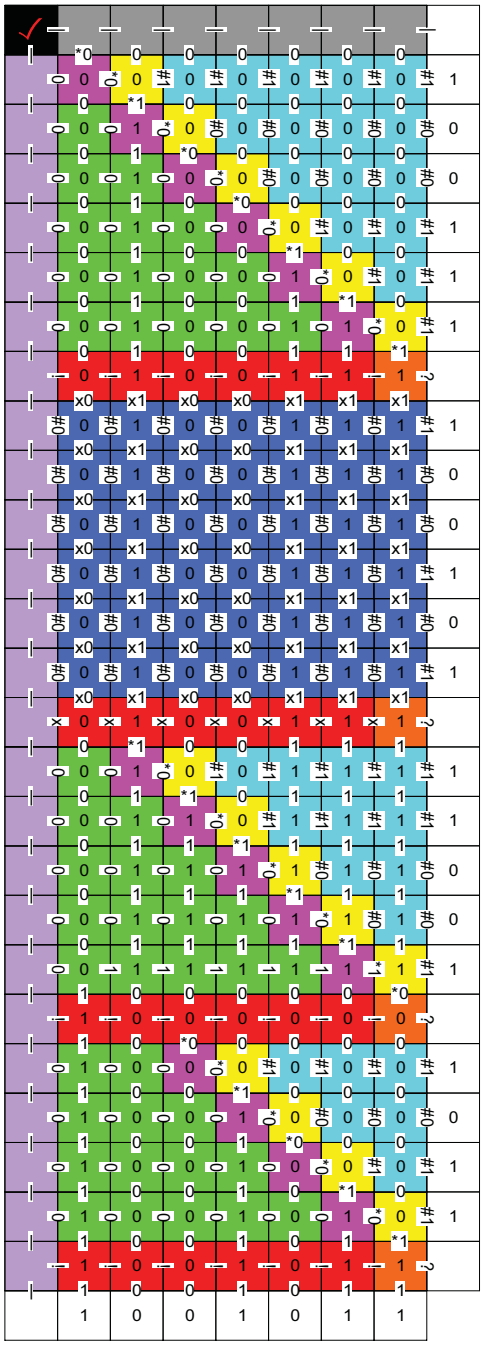

Figure 3. An example execution of the tile system that solves SubsetSum. The clear tiles encode the input: a set of numbers: $\{11 = 1011_2, 25 = 11001_2, 37 = 100101_2, 39 = 100111_2\}$ along the right column, and a target number $75 = 1001011_2$ along the bottom row. Because $75 = 11 + 25 + 39$, one nondeterministic execution of the tile system finds the proper selection of numbers and attaches the special ✓ tile. If there were no subset of numbers whose sum equaled 75, no such tile could attach.

variable (shown in the center of each tile) and four *interfaces*, i.e. side variables (shown on the sides of each tile). The *topology* is a 2-D grid of components that allows neighbors on the grid to interact. The components exhibit two *behaviors*: cooperating with neighbors to recruit suitable new components to attach, as described in Section 3.2.2, and reporting the solution to the user. Recruitment is the principal functionality performed by a given tile. The *interaction* consists of exchanging data about a component's sides in order to recruit. The *data flow* is limited to the components' state variable and sides, allowing components to tell their neighbors their state and their side labels, but no other information.

3.2.1 Replication

The tile system from Section 3.1.2 that solves SubsetSum has 49 distinct computational tile types and 7 seed tile types. At the start of the computation, each node on the network is assigned to represent a particular type of a computation tile and a particular type of a seed tile. The simplest process that accomplishes this is each node randomly deciding to represent a particular tile type from a list of all types of tiles. We should note that there are simple mechanisms for assigning nodes to tile types without sharing the complete list of tiles with any one node. We do not go into the details of those mechanisms in this paper, and just allow the nodes to select a type from the list.

The client sets up a single seed on the network, as described in Section 3.3. Each tile knows of its neighbors. The seed tiles then replicate twice, to create two additional copies of the seed on the network. To do so, each tile finds another tile on the network of the same type as itself, and designates it to represent part of the seed. It also coordinates with its neighbors to inform the new copy of its new neighbors. After creating two copies of the seed, the tiles begin the recruitment process. The newly created seeds will also each replicate twice, thus creating a number of seeds exponential in time. The seeds continue to replicate and self-assemble until one of the assemblies finds the solution, at which time the client broadcasts a signal to cease computation, and the replication and recruitment stop.

Note that we do not know of algorithms to solve NP-complete problems that do not require an exponential number of parallel executions, thus every fixed-size network can be overwhelmed by a large enough input. A goal of this architectural style is to distribute the computation across many physical nodes to execute in parallel, but for a given network size and input size, one can set bounds on the number of components deployed on each physical node to prevent overloading those nodes (each node can determine its own resources and stop accepting recruitment and replication requests if it is overwhelmed). We are in the process of

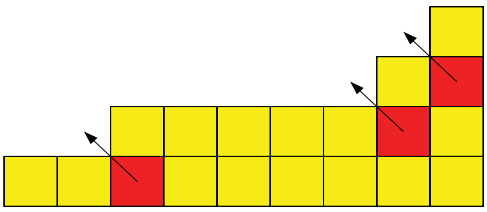

Figure 4. Tiles that have a north and west neighbor (red tiles) can recruit new tiles to attach to their northwest. The arrows indicate where the new recruited tiles would attach.

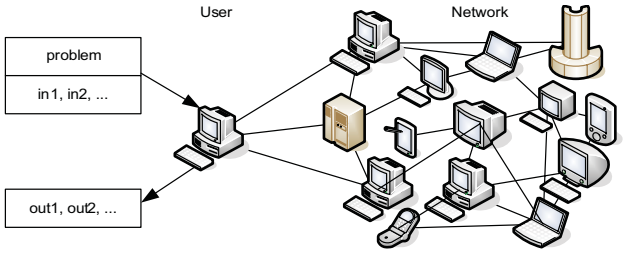

Figure 5. A schematic of a system implementing a tile style architecture.

investigating the practical implications of this observation on actual networks.

3.2.2 Recruitment

In a temperature 2 system (such as the system described in Section 3.1.2 that solves SubsetSum), a tile that has north and west neighbors recruits a new tile to attach to its northwest. Figure 4 indicates several places in a sample assembly where tiles are ready to recruit new tiles. The arrows indicate where the new recruited tiles would attach. A recruiting tile (a red tile in Figure 4) asks its north and west neighbors for their west and north side labels, respectively, and then finds another node on the network whose side labels match and informs the new node of its new south and east neighbors.

3.3 Using the Tile Style

A user who wishes to solve a computationally intensive and easily parallelizable problem, e.g. an NP problem, and has access to a large unreliable network, may use the tile architectural style to design a system to solve her problem. Figure 5 shows a high-level schematic of the interaction between the user's computer and the network.

The user has two options: use the tile style to design her own architecture based on the tile system that solves her particular problem, as we describe in [7], or reduce her problem to SubsetSum, using a standard polynomial time reduction [18], and use the SubsetSum tile system. Whichever tile system the user chooses will serve as the template for the architecture, with the system's tiles defining the types of components. Part of a tile system is the description of seeds that encode inputs (e.g., Figure 1(b) shows the seed for adding 10 and 11, and the clear tiles in Figure 3 are the seed for that SubsetSum computation). The user sets up a seed to encode her input and assigns computers on the network to represent the seed tiles. Once the initialization is complete, starting with the seed tiles, adjacent components deploy on other network nodes to represent fitting components and eventually produce the solution. The solution tiles (the ✓ tile for the SubsetSum system) then report their state to the user.

3.4 Answering SubsetSum in the Negative

We have described how an assembly that finds the proper subset of numbers that adds up to the target number reports the success to the client computer. However, deciding that no subset of the given numbers adds up to the target number is more difficult. No assembly can ever claim to have found the proof that no such subset exists. Rather, the absence of assemblies that have found such a subset stands to provide some certainty that no such subset exists. Because, for an input of size n, there are 2^n possible subsets, if 2^n assemblies find no suitable subset, then the client knows there does not exist such a subset with probability at least $\left(1 - e^{-1}\right)$. After exploring 2×2^n assemblies (this takes twice as long as exploring 2^n assemblies), the probability grows to at least $\left(1 - e^{-2}\right)$. After $m \times 2^n$ assemblies, the probability is at least $(1 - e^{-m})$. Thus as time grows linearly, the probability of error diminishes exponentially. Given the network size and bandwidth, it is possible to determine how long one must wait to get the probability of an error arbitrarily low. Further, if desired, it is possible to design a scalable reporting mechanism for counting exactly how many assemblies have attempted the computation and bound the probability more exactly.

3.5 Efficiency of the Tile Style

The tile style is particularly aimed at large networks. When a client wishes to solve a highly parallelizable problem and needs discreetness, she may choose to do so simply on her own single computer, perhaps on a small private network of trustworthy computers, or using the tile style on a large insecure network. The disadvantage of computing on a network is that communication between components can be as much as 1000 times slower over a network than it is between components on a single computer. The com-

putation is further slowed down by the fact that tiles may have to perform more basic operations than a program that is not restricted by the components of the tile style (e.g., the example in Figure 3 uses 242 tiles, whereas a simpler program could just add the three 6-bit numbers using eighteen bit operations). The slowdown due to the use of tiles and use of a network is linear in the input size, although the constant factor may be large (in the above example, as large as $1000 \times \frac{242}{18} \approx 13000$). The upside is that the tile style distributes the work over the network, and allows computation to happen in parallel. Thus the network simply needs to be large enough to sufficiently amortize the cost of using the tile style (for the above example, the network needs at least 13000 nodes).

4 Evaluation

In this section, we present our preliminary theoretical analysis of discreetness, fault-tolerance, and scalability of the tile style based systems.

4.1 Discreetness

We call a distributed system *discreet* if, with high probability, for all time, for all nodes on the network, each node cannot figure out the input to the algorithm it is executing.

The tile system from Section 3.1.2 that solves SubsetSum has 49 distinct computational tile types and 7 seed tile types. Each tile type encodes no more than two bits of the input (one bit of the target number and one bit of one of the sum numbers). The ✓ tile encodes the solution, but has no knowledge of the input.

If every tile in the assembly were represented by a different node on the network, it would be trivial to argue that the computation were discreet; however, since a single node on the network may represent several tiles, the argument has to take into account the fact that the node is not aware of its location in the assembly, and thus it does not know the location of the bits of input.

Therefore, every node on the network may be aware of either some bits of the input or the solution, but not both. Further, of the bits the node knows, it does not know their position, thus the information available to any one node is highly limited, resulting in a discreet system.

4.2 Fault-Tolerance

We call a distributed system *fault-tolerant* if, given a fraction of the network nodes failing or acting in a malicious fashion, the probability of successful computation can still be bounded arbitrarily close to 1 without paying a major (exponential) cost in speed. Note that usually, fault-tolerance is defined only in terms of faulty nodes; however, the tile style allows our definition to be even stronger, disallowing either faulty or malicious nodes from breaking the computation.

While the system from Section 3.1.2 that solves SubsetSum does not have the fault-tolerance properties we desire, we rely on related work in fault-tolerance of tile systems to show that we could design a fault-tolerant sibling of that tile system. Winfree et al. [21] and Goel et al. [9] have shown that given a tile system and a certain fraction of malicious tiles, one can linearly slow down the system by increasing the number of tiles (e.g., break each tile into a 2×2 grid and represent it with four tiles), and bring the probability of error exponentially close to 0. For example, increasing the number of tiles and the computation speed by $O(n)$ would bring the previous error probability of e to $e^{O(n)}$.

4.3 Scalability

We call a distributed system *scalable* if the rate of communication per node does not grow with the input size.

Every tile in the assembly requires a constant amount of communication to attach to the assembly. Once attached, it can only participate in the recruiting of two other tiles, thus the communication associated with each tile is bounded. Because there is only a constant number of types of tiles (49 for SubsetSum), the number of nodes each component has to sample to recruit the appropriate component is bounded by a constant, with high probability. For example, for the tile system described in Section 3.1.2 that solves SubsetSum, a tile has to sample 49 other nodes to have at least an $1 - e^{-1}$ probability of finding a fitting component. Sampling $49 \times k = 98$ nodes brings the probability of success to $1 - e^{-k}$. Thus with high probability, no node representing a component will need more than a constant amount of communication originating from it in order to recruit another component to attach.

5 Contributions

We have developed an architectural style for discreet, fault-tolerant, and scalable computation on a large network and have argued how to design systems based on this architectural style for every computational problem solvable on a computer. We have further presented an architecture that solves SubsetSum, an NP-complete problem. Because of the nature of NP-complete problems, this architecture can be used to solve all NP problems, such as the traveling salesman problem, a variety of scheduling problems, resource allocation problems, and boolean formula satisfiability problems such as 3-SAT.

The architectural style is based heavily on the theoretical study of self-assembly, and systems built based on the style inherit the discreetness, fault-tolerance, and scalability

of tile systems. We have argued that the system designed based on the tile style and that solves SubsetSum, is discreet, fault-tolerant, and scalable, and further that these are provable properties of the system, which is a desirable quality of software architectures.

6 Acknowledgement

This work is sponsored in part by the National Science Foundation under Grant number ITR-0312780.

References

[1] L. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266:1021–1024, 1994.

[2] L. Adleman, Q. Cheng, A. Goel, M.-D. Huang, and H. Wasserman. Linear self-assemblies: Equilibria, entropy, and convergence rates. In *Proceedings of the 6th International Conference on Difference Equations and Applications (ICDEA 2001)*, Augsburg, Germany, June 2001.

[3] L. Adleman, A. Goel, M.-D. Huang, and P. M. de Espanes. Running time and program size for selfassembled squares. In *ACM Symposium on Theory of Computing (STOC02)*, pages 740–748, Montreal, Quebec, Canada, 2001.

[4] L. Adleman, J. Kari, L. Kari, and D. Reishus. On the decidability of self-assembly of infinite ribbons. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS'02)*, pages 530–537, Ottawa, Ontario, Canada, November 2002.

[5] R. Barish, P. W. K. Rothemund, and E. Winfree. Two computational primitives for algorithmic self-assembly: Copying and counting. *Nano Letters*, 5(12):2586–2592, 2005.

[6] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[7] Y. Brun. Arithmetic computation in the tile assembly model: Addition and multiplication. *Theoretical Computer Science*, 10.1016/j.tcs.2006.10.025, 2006.

[8] Y. Brun. Solving NP-complete problems in the tile assembly model. Technical Report USC-CSSE-2007-703, Center for Software Engineering, University of Southern California, 2007.

[9] H.-L. Chen and A. Goel. Error free self-assembly with error prone tiles. In *Proceedings of the 10th International Meeting on DNA Based Computers*, Milan, Italy, June 2004.

[10] P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[11] P. T. Devanbu and S. Stubblebine. *Software Engineering for Security: A Roadmap*, pages 225–239. ACM Press, 2000.

[12] T. J. Fu and N. C. Seeman. DNA double-crossover molecules. *Biochemistry*, 32(13):3211–3220, 1993.

[13] M. Mikic-Rakic, N. R. Mehta, and N. Medvidovic. Architectural style requirements for self-healing systems. In *Proceedings of First Workshop on Self-Healing Systems*, Charleston, SC, Nov. 2002.

[14] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[15] P. W. K. Rothemund, N. Papadakis, and E. Winfree. Algorithmic self-assembly of DNA Sierpinski triangles. *PLoS Biology*, 2(12):e424, 2004.

[16] P. W. K. Rothemund and E. Winfree. The program-size complexity of self-assembled squares. In *ACM Symposium on Theory of Computing (STOC02)*, pages 459–468, Montreal, Quebec, Canada, 2001.

[17] M. Shaw and D. Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

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

[19] E. Winfree. *Algorithmic Self-Assembly of DNA*. PhD thesis, California Insitute of Technology, Pasadena, CA, June 1998.

[20] E. Winfree. Simulations of computing by self-assembly of DNA. Technical Report CS-TR:1998:22, California Insitute of Technology, Pasadena, CA, 1998.

[21] E. Winfree and R. Bekbolatov. Proofreading tile sets: Error correction for algorithmic self-assembly. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS02)*, volume 2943, pages 126–144, Madison, WI, June 2003.