Fault and Adversary Tolerance as an Emergent Property of Distributed Systems' Software Architectures

Yuriy Brun and Nenad Medvidovic
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781, USA
{ybrun, neno}@usc.edu

ABSTRACT

Fault and adversary tolerance have become not only desirable but required properties of software systems because mission-critical systems are commonly distributed on large networks of insecure nodes. In this paper, we describe how the tile style, an architectural style designed to distribute computation, can inject fault and adversary tolerance. The result is a notion of tolerance that is entirely abstracted away from the functional properties of the software system. The client may specify what fraction of the network is faulty or malicious (e.g., 25%) and the acceptable system failure rate (e.g., 2^{-10}), and the system's architecture adjusts automatically to ensure a failure rate no higher than the one specified. The technique is entirely automated and consists of a "smart redundancy" mechanism that brings the failure rate exponentially close to 0 by slowing down the execution speed linearly.

1. INTRODUCTION

With the growth of distributed systems, fault tolerance has advanced from being a desired non-functional property to an absolute requirement for system stability. Software architecture had been identified as one approach to ensuring fault tolerance in a system. Traditionally, there have been two, perhaps complimentary, methods to providing fault tolerance within an architecture: (1) with the support of a fault-tolerant component whose job is to oversee the functional components of a system, detecting and correcting faults, and (2) by requiring the individual components and connectors of the system to themselves be fault-tolerant. More recently, a third method has become evident: providing fault tolerance as an emergent property of the software architecture. In contrast to the former two, this third method requires no particular component to take care of its own fault tolerance nor of the fault tolerance of the entire system, but the component and connector interactions ensure that the system recovers seamlessly from failures.

Fault tolerance as an emergent property of the software architecture is in some ways preferable to alternate methods because it requires no design and implementation of separate components or connectors. The cost, however, may include a more complex than otherwise necessary architecture. One important question becomes what is the trade-off between complexity of the architecture and complexity of the components and connectors of that architecture to ensure fault tolerance, and whether placing all the burden on the architecture is an acceptable price to pay for large distributed systems. In this paper, we present the tile style, an architectural style that ensures fault tolerance as an emergent behavior and argue that this style carries a low cost.

With the growth of the Internet as a computational medium, distributed software systems have begun to require not only fault tolerance but also adversary tolerance: the ability to cope with powerful and malicious network nodes attempting to mislead or break computations, as well as learn computation's private and sensitive data. As with fault tolerance, adversary tolerance may be implemented in a system via a single component, within each component and connector, or as an emergent property of the architecture. The tile style, the architectural style we present in this paper, provides adversary tolerance as an emergent behavior without any additional cost over providing fault tolerance.

Let us explore example scenarios to which the tile architectural style can be applied to ensure fault and adversary tolerance. The Human Genome Project processed a tremendous amount of data and required the collaboration of hundreds, if not thousands, laboratories around the world. A single laboratory could have misdirected the project by providing bad data, leading to only a partially correct genome sequence. Other projects may be more sensitive to failure, where a single faulty or malicious node may derail the computation in a way that will result in a fully incorrect output. Solving a problem such as that of the Human Genome Project in a distributed software system would require a high level of complexity. It has been shown that such systems are most effectively approached from a software architectural perspective (e.g., [17]). In particular, architectural *styles* [21] present generic design solutions that can be applied to problems with shared characteristics.

Biological systems often exhibit properties such as fault and adversary tolerance, and surpass those properties in complex human-engineered systems. For example, the human being is an immensely more complex system, with orders of magnitude more components and more behaviors, than today's most complex engineered systems. And still, human beings are more resilient to failures of individual components and injections of malicious bacteria and viruses than engineered software systems are to component failure and computer virus infection. Other biological systems, for example worms and sea stars, are capable of recovering from such serious hardware failures as being cut in half (both worms and sea stars are capable of regrowing the missing pieces to form two nearly identical organisms), yet we envision neither a functioning desktop, half of which was crushed by a car, nor a machine that can recover from being installed with only half of an operating system.

EFTS'07 September 4, 2007, Dubrovnik, Croatia

In some ways, the tile architectural style is an architecture of the biological systems, which brings forward many of the fault and adversary tolerant features of biology and allows software systems to benefit from nature's achievements. More formally, the tile style is based on the tile assembly model [19], a formal mathematical study of self-assembly, which attempts to encompass nature's ability to self-assemble with a formal mathematical model.

In order to produce software systems based on the tile assembly model, we have developed and analyzed systems that compute complex functions within that model. In particular, we have constructed tile based-systems that solve NP-complete problems [7]. We have then proposed the tile style, an architectural style based on the tile assembly model, and argued briefly that it provides properties of discreetness, fault tolerance, and scalability [8, 9]. In this paper, we leverage this work and explore the properties of fault and adversary tolerance, which emerge from using the tile style without explicit engineering or designing of fault or adversary-tolerant components or connectors. We propose and begin the analysis of the tile style, while a more complete analysis, including empirical evidence remain future work.

In a sense, the tolerance behaviors come for free from the tile style, though a closer analysis reveals that there is a slight overhead evident in a linear slowdown of the system, as well as, of course, somewhat of a learning curve to using the tile style in the first place, though the latter cost is a necessity for every new architectural style.

The rest of this paper is structured as follows: Section 2 will discuss work in software architectures and fault tolerance that is related to our work. Section 3 will discuss some relevant details and intuition of the tile architectural style, although we refer the reader to [9] for a more complete tile style definition. Section 4 will describe how the tile style provides emergent fault and adversary tolerance. Finally, Section 5 will summarize our contributions.

2. RELATED WORK

In this section, we describe related work in software architectures and in fault and adversary tolerance.

2.1 Software Architectures

Software architecture has been identified as an important part of building almost all large systems [17]. 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 [16]. However, in our proposed tile 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 [12]. Accordingly, the tile style, in principle, 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., [3, 11, 21]), we directly leverage Mikic-Rakic et al.'s [16] definition in formulating the tile style. Mikic-Rakic et al. 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 the tile architectural style in Section 3.

2.2 Fault and Adversary Tolerance

Gärtner has attempted to structure the field of fault tolerance by formalizing and standardizing the definitions used in the field [13]. He separates the process of fault tolerance into two phases: *detection* and *correction*. The tile architectural style presented in this paper performs both phases simultaneously, much in the same way coding theory has allowed one to encode data in a way that if part of that data is lost or corrupted, the errors can be detected and corrected. Gärtner argues that redundancy is necessary for fault tolerance. Coding theory uses redundancy to allow integrated detection and correction of errors. Similarly, the tile style uses an efficient notion of redundancy to detect and correct errors.

Seo et al. have argued that an effective way to ensure fault tolerance in complex software systems is to employ the principles of software architectures [20]. They also argue that the complexity of pervasive software systems creates challenges in maintaining the desired fault tolerance properties and that software architectures, in some way, can help create an abstraction barrier between the functional properties of a software system and the fault tolerance ensuring components. A key aspect of the tile style is that fault and adversary tolerance are emergent properties of the architectural style, and once a system's architecture is designed using the tile architectural style, the architecture itself takes care of the redundancy necessary to ensure tolerance automatically. Thus, the tile style provides the ultimate abstraction barrier between functional properties of the system and the tolerance aspects of the system.

3. TILE ARCHITECTURAL STYLE

This section describes the tile style, parts of which have been previously introduced in [5, 8, 9], but we repeat that information here for completeness.

Adleman first proposed solving NP problems using DNA [1]. Rothemund and 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 [19]. The tile assembly model is a model of crystal growth, in which 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. The tile assembly model has been shown to be Turing universal [2, 23]. The tile assembly model is a computational model that is somewhat similar to cellular automata, but instead of being able to switch state, the individual square "automata" (called "tiles" precisely because they cannot change state) attach to other tiles following simple matching rules. We have extended their work to explore efficient computation within the tile assembly model [4, 6, 7].

In [8, 9], we explain how one can design efficient tile systems to solve computational problems. In particular, we describe a system to solve $SubsetSum$, an NP-complete problem (a formal description of this system originally appeared in [7]). While we refer the reader to [9] for a full description of the tile style, we will present an example of solving a $SubsetSum$ problem to ground the reader.

Figure 1 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. The system encodes the input in binary using the white tiles along the bottom row and right-most column, e.g., 75 = 1001011_2. The colorful tiles then self-assemble (attach when their sides match) to the white tiles. Because $75 = 11 + 25 + 39$, one nondeterministic execution of the tile system finds the proper selection of numbers and attaches the special ✓ tile in the top left corner. If there were no subset of numbers whose sum equaled 75, no such tile could attach. We refer the reader to [7] for a formal proof of correctness of the system that solves $SubsetSum$.

The basic idea of the tile style is to have individual nodes on a network represent the tiles, and communicate with each other to self-assemble, as shown in Figure 1, to solve NP-complete problems. There are a number of details to the tile style, including algorithms to discover nodes deploying particular tile types and to replicate input seeds to perform nondeterministic computation, which we do not describe here. A complete description of the tile style can be found in [9]. For the purposes of this paper, it is sufficient for the reader to understand that nodes on a network can represent tiles to perform arbitrary computations (because the tile assembly model has been shown universal), and in particular, solve NP-complete problems efficiently.

Finally, we describe how a client would use the tile style. The tile style is an architectural style for the architecture of a system that distributes computation on a large network. Given a computational problem, there are three ways to use the tile style to solve that problem. The most complex way is to "write a tile program" (design a set of tiles) to solve your problem and then use the tile style to design the architecture of a system based on that tile program. This approach requires programming with tiles, and while that process is not unlike traditional computer programming (one can design abstractions, subroutines, etc.), the learning curve and the lack of proper design tools may make this process inefficient and error-prone. A simpler approach arises from the fact that one can write a compiler that takes an arbitrary universal language program, e.g., a Java program, and compiles it into a set of tiles. While no such compiler exists at the moment, the theory behind building one has been worked out, in part in [23], and in part in our heads. This approach would open the tile style to a wide variety of problems; however, it is unlikely to be as efficient as designing your own tile system to solve a problem because we do not know of efficient ways to parallelize code, in the general case. Finally, we recommend the third approach, which consists of translating your problem to one with a known tile system solution and using that tile style system to solve your problem. In particular, given an NP-complete problem[1] a client wishes to solve, she can translate that problem to $SubsetSum$ via a polynomial-time reduction, and then solve $SubsetSum$ using the system from Figure 1 and described in [7]. The third approach is only applicable to NP-complete problems, which is a large and important class of computationally intensive problems, and has the great advantage that the client never has to program using tiles.

We have argued in [9] that the tile style allows distributing information discreetly (without telling individual nodes on the network the algorithm or the input to the computation) and that the system is scalable. We also allude to possible properties of fault and adversary tolerance, which we explore further and more formally now. The notions of the complexities of tile computation are interesting

[1]We have demonstrated how the tile style can be used to solve all NP-complete problems via polynomial-time reductions [9]. However, the technique extends directly to problems that are P-SPACE-complete, and other complexity classes.

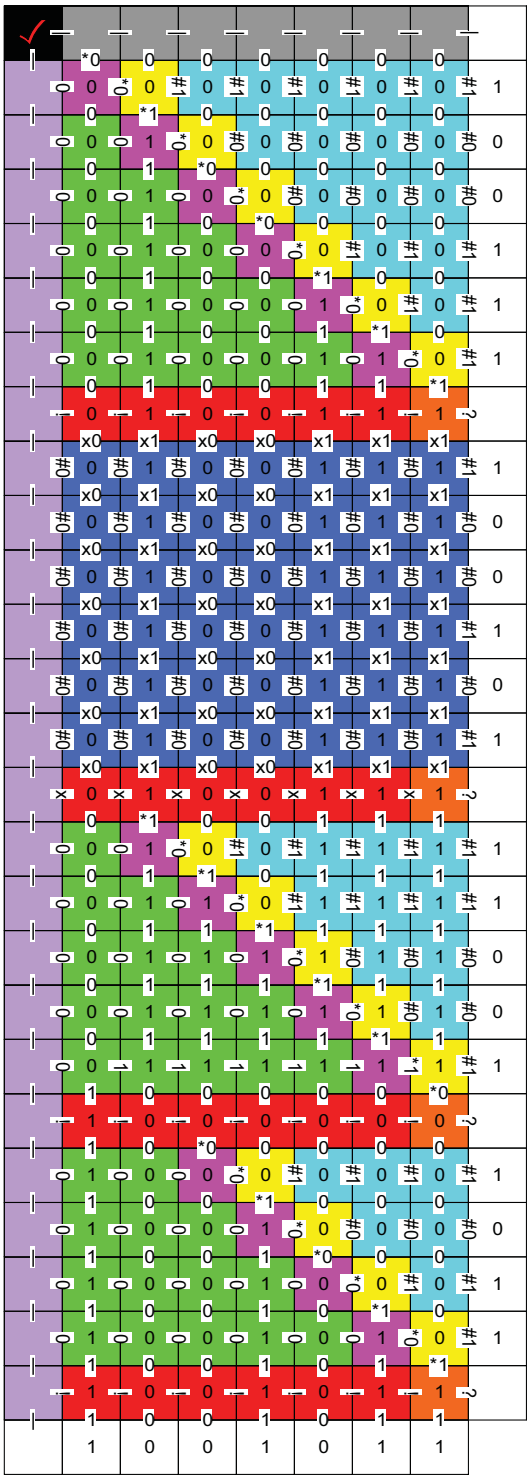

Figure 1: An example execution of the tile system that solves $SubsetSum$. The clear tiles encode the input: a set of numbers: $\{11 = 1011_2, 25 = 11001_2, 37 = 100101_2, 39 = 100111_2\}$ along the right column, and a target number $75 = 1001011_2$ along the bottom row. Because $75 = 11 + 25 + 39$, 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.

and clearly integral to this paper. We refer the reader to [7] and [9] for descriptions of these complexities.

4. EMERGENT TOLERANCE

The tile style abstracts the properties of fault and adversary tolerance away from the computational aspects of the system. Our goal is a system that does not require the designer to put effort into making the system fault or adversary-tolerant. Once designed using the tile style, a system's architecture allows the designer to specify two parameters: the fraction of nodes on the network that may be faulty or malicious, and the acceptable rate of system failure. For example, a designer may say that some fraction of the network is malicious,[2] but the client can only accept a failure rate of 2^{-10}. The system then automatically self-adapts to produce the proper failure rates, without the designer, or anyone else, having to write code.

First, we will give our definition of fault and adversary tolerance in Section 4.1. We will then provide an intuitive explanation of how the tile style achieves the fault and adversary tolerance properties in Section 4.2. The intuition is not entirely accurate, but goes a long way to making the technique more understandable. Then, we will more formally explain the tile style's emergent tolerance and how we leverages work on error-correction within the tile assembly model in Section 4.3. Finally, we will discuss the types of faults and attacks the tile style tolerates in Section 4.4.

4.1 Definition

We call a distributed system *fault-tolerant* (adversary-tolerant) if, given a fraction of the network nodes failing (acting in a malicious fashion), the probability of successful computation can still be bounded arbitrarily close to 1 without paying an exponential cost in speed. In particular, systems designed using the tile style allow the architect to slow down the system linearly while lowering the failure rates exponentially.

4.2 Intuition

Computer science has long used redundancy to induce fault tolerance in systems. The basic idea is that if a component has a probability p of failure, if the failures are independent, two components performing the same task have only a p^2 probability of failure. The above calculation depends heavily on the assumption of independence of failures. In our realm, this assumption implies that a node's failure must have no dependence on another node's failure, which can be achieved by ensuring that either the implementation of the components or the attacks on them differ.

If the failure is a *crash* failure (component either returns the correct answer or crashes, returning no answer), then k components performing the same tasks reduces the probability of failure of the entire system to p^k. A similar result can be achieved for *Byzantine* failures (component either returns the correct answer or an incorrect answer but no indication that it has failed) by employing a system of voting. In this case, $\Theta(k)$ components are necessary to perform the same task to reduce the probability of failure of the entire system to p^k. In essence, the tile style can employ a redundancy approach. In the "basic" tile style, each tile component of the assembly is deployed on some network node. If that node is malicious, or faulty, it may attach incorrectly or attempt to recruit other tile components incorrectly and break the entire computation.

[2]Our approach requires an estimate of an upper bound on the fraction of faulty or malicious nodes. Calculating such a bound depends on the nature of the system, as the faults may be related to hardware or environmental factors. While the bound needs only to be an estimate, the guarantees provided by the tile style will only be as accurate as this estimate.

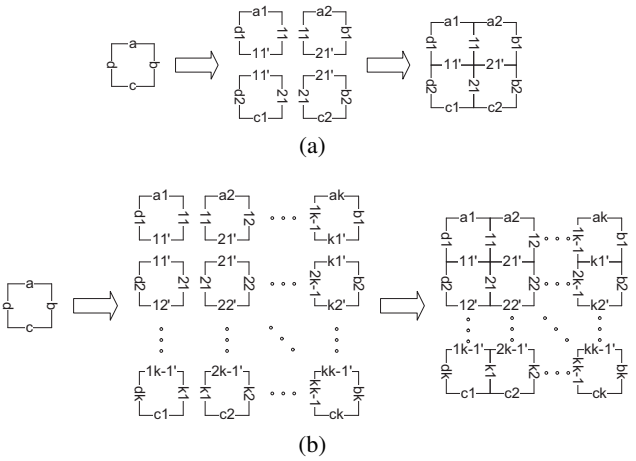

Figure 2: A 2×2 smart redundancy mechanism (a) transforms each tile type in a system into four tile types, such that the four can attach uniquely to each other. In general, a $k \times k$ smart redundancy mechanism (b) transforms each tile type in a system into k^2 tile types that can all uniquely attach to each other.

It is possible to have multiple nodes be responsible for deploying each tile, checking each others' computations, and thus correcting crash errors and voting to avoid Byzantine errors.

4.3 Error-Correction

While direct redundancy is one way to accomplish the goal of fault tolerance, it is possible to use nodes more wisely to correct each other by exploring the field of error-correction in self-assembly.

Winfree et al. [25] and Reif et al. [18] have shown that given a tile system and a certain fraction of malicious tiles, one can increase 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 by a constant factor c, from k to ck in [25], (or from k to ck^2 in [18]) would bring the previous error probability of ϵ to $\epsilon^{O(c)}$. In some sense, they have developed "smart redundancy."

Figure 2(a) shows an example of a tile being represented by four tiles, arranged in a 2×2 square, with unique internal binding domains. The new 2×2 square contains essentially all the information the original tile contained. It follows that if all tile types are transformed as we described, the assemblies made by the new system will be four times larger (two times larger in each of the two dimensions) than the original system's assemblies, but will otherwise encode the same information. Note that this transformation can be done entirely automatically: given a tile system $\mathbb{S}$ with n tiles, one can create another tile system $\mathbb{S}'$ with $4n$ tiles that performs the same computation.

A tile can be represented by a $k \times k$ square, in a similar fashion, and shown in Figure 2(b) and described in [25]. By transforming n tile types into k^2n tile types, the $k \times k$ smart redundancy mechanism linearly increases the size of the assemblies (an n tile assembly becomes a k^2n tile assembly), while raising the probability of a growth error to the power k. For the complete details and the formal proofs of the error rate improvements, we refer the reader to [25].

By applying tile assembly model error-correction techniques to the tile style, we have begun a new exploration of these techniques because software faults differ from the faults traditionally examined

in the study of self-assembly. This exploration is leading to ways of classifying the techniques, as well as modifying them to be robust to a larger class of faults.

Several other researchers have done work on error-correction in the tile assembly model [10, 18, 22, 24], looking at increasing accuracy without increasing the assembly size, and healing *catastrophic errors* such as the *death* of a large portion of the assembly at once. This work may be useful for errors such as large portions of the network failing at once and massive attacks or failures that target geographically close tiles, but the exact exploration of that area is beyond the scope of this paper.

4.4 Fault Scenario Analysis

The error-correction work proposed in [18, 25] applies directly to tile assembly model systems. As the tile assembly model is a model of biological systems, these error-correction techniques safeguard against errors commonly found in biology (e.g., what are called *nucleation* and *growth* errors). A tile system that employs such error-correction techniques may be resilient to mismatched tiles attaching in the wrong places, which is an accurate model of incorrect molecular attachment. However, the tile architectural style allows computers to represent tiles, and the types of errors that may occur are likely to differ from the biological variety. While it is not too difficult to show that these error-correction techniques, applied to the tile architectural style, would correct some particular types of faults, the key aspect of this work is showing that neither faulty nor malicious nodes on a network can break these particular error-correction techniques. (It may be worthwhile to note that some other tile assembly model error-correction techniques described in [10, 18, 22, 24] would not be as helpful for the tile style as the ones we have selected.)

By applying the error-correction mechanism from [18, 25] to the tile style, we foresee countering such attacks as nodes pretending to host foreign tile components, attempting to report incorrect computation results, and overwhelming the client or other nodes on the network with traffic. Theorem 4.1 summarizes that result.

THEOREM 4.1. *Let $\mathbb{T}$ be a computational tile system that computes the function f, and let $\mathbb{S}$ be the software system with the architecture derived by the tile style from $\mathbb{T}$. Let ρ be the probability of $\mathbb{S}$ failing on network N with some faulty and some malicious nodes. Then for all $j \geq 2$, it is possible to design a tile system $\mathbb{T}'$ that also computes f such that the software system with the architecture derived by the tile style from $\mathbb{T}'$ has the probability of failing on network N of ρ^j.*

The proof of Theorem 4.1 is a combination of proofs of theorems in [18, 25], the fact that each tile component does not know its location in the assembly, and the observation that when a tile is represented by a $k \times k$ block of tiles, as long as there are fewer than k improper tiles, that block either assembles completely correctly or will not complete. Theorem 4.1 is a stronger statement than the theorems in [18, 25] because those theorems assume only the presence of tiles allowed by the tile system, whereas we assume all possible tiles, as well as the presence of tiles conspiring together to break the assembly. While we outline the proof here, it remains part of our future work to formally prove Theorem 4.1, as well as empirically verify this result.

The implication of Theorem 4.1 is that given a software system designed using the tile style, and a network with faulty or malicious nodes, if the system has a failure rate of 25%, by using smart redundancy as described in Figure 2(b) with $k = 5$, the probability of the system failing decreases to 2^{-10}, while slowing down the execution speed only by a factor of 5, because $\left(\frac{1}{4}\right)^5 = 2^{-10}$.

5. CONTRIBUTIONS

We have previously argued that the tile style is an architectural style for distributing computation on a large network in a discreet, efficient, and scalable manner. Here, we have explored the properties of fault and adversary tolerance, which are emergent properties of the tile style. The designer need not worry about fault and adversary tolerance when designing the system. As long as she uses a tile style architecture, she can specify the fraction of nodes that are faulty or malicious (e.g., 25%) and the acceptable system failure rate (e.g., 2^{-10}), and the system's architecture adjusts automatically to guarantee that failure rate via a smart redundancy mechanism.

While we have provided formal arguments for the tile style's discreetness, efficiency, scalability, and now fault and adversary tolerance, it remains our future work to verify these properties empirically. We are currently in the process of developing a distributed computing platform for implementing and deploying tile-style architectures. To this end, we are leveraging Prism-MW, our light-weight architectural middleware platform [14], as well as the lessons learned from NASA JPL's OODT grid platform in whose construction we participated [15]. Once complete, the resulting middleware will allow us to deploy tile style-based systems on virtual and real networks, study their properties, and provide specific optimizations to account for real-world scenarios.

6. ACKNOWLEDGMENTS

This work is sponsored in part by the National Science Foundation under Grant number ITR-0312780. The authors wish to acknowledge Leonard Adleman, Dusting Reishus, and David Woollard for helpful discussions of this work. Finally, the authors wish to thank the anonymous reviewers for their insightful comments and suggestions.

7. REFERENCES

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

[2] L. Adleman, J. Kari, L. Kari, and D. D. Reishus. On the decidability of self-assembly of infinite ribbons. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS02)*, pages 530–537, Ottawa, Ontario, Canada, November 2002.

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

[4] Y. Brun. Arithmetic computation in the tile assembly model: Addition and multiplication. *Theoretical Computer Science*, 378:17–31, June 2006.

[5] Y. Brun. A discreet, fault-tolerant, and scalable software architectural style for internet-sized networks. In *Proceedings of the Doctoral Symposium at the 29th International Conference on Software Engineering (ICSE07)*, pages 83–84, Minneapolis, MN, USA, May 2007.

[6] Y. Brun. Nondeterministic polynomial time factoring in the tile assembly model. Technical Report USC-CSSE-2007-707, Center for Software Engineering, University of Southern California, 2007.

[7] 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.

[8] Y. Brun and N. Medvidovic. An architectural style for solving computationally intensive problems on large

networks. In *Proceedings of Software Engineering for Adaptive and Self-Managing Systems (SEAMS07)*, Minneapolis, MN, USA, May 2007.

[9] Y. Brun and N. Medvidovic. Discreetly distributing computation via self-assembly. Technical Report USC-CSSE-2007-714, Center for Software Engineering, University of Southern California, 2007.

[10] 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 (DNA04)*, Milan, Italy, June 2004.

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

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

[13] F. C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1):1–26, 1999.

[14] S. Malek, M. Mikic-Rakic, and N. Medvidovic. A style-aware architectural middleware for resource-constrained, distributed systems. *IEEE Transactions on Software Engineering*, 31(3):256–272, 2005.

[15] C. A. Mattmann, D. J. Crichton, N. Medvidovic, and S. Hughes. A software architecture-based framework for highly distributed and data intensive scientific applications. In *Proceedings of the 28th international conference on Software engineering (ICSE06)*, pages 721–730, Shanghai, China, 2006. ACM Press.

[16] M. Mikic-Rakic, N. R. Mehta, and N. Medvidovic. Architectural style requirements for self-healing systems. In *Proceedings of 1st Workshop on Self-Healing Systems*, Charleston, SC, USA, November 2002.

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

[18] J. H. Reif, S. Sahu, and P. Yin. Compact error-resilient computational DNA tiling assemblies. In *Proceedings of the 10th International Meeting on DNA Based Computers (DNA04)*, Milan, Italy, June 2004.

[19] P. W. K. Rothemund and E. Winfree. The program-size complexity of self-assembled squares. In *Proceedings of the ACM Symposium on Theory of Computing (STOC00)*, pages 459–468, Portland, OR, USA, May 2000.

[20] C. Seo, S. Malek, G. Edwards, N. Medvidovic, B. Petrus, and S. Ravula. Exploring the role of software architecture in dynamic and fault tolerant pervasive systems. In *Proceedings of the Workshop on Software Engineering of Pervasive Computing Applications, Systems and Environments (SEPCASE07)*, Minneapolis, MN, USA, May 2007.

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

[22] D. Soloveichik and E. Winfree. Complexity of compact proofreading for self-assembled patterns. *Lecture Notes in Computer Science*, 3892:305–324, 2006.

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

[24] E. Winfree. Self-healing tile sets. *Nanotechnology: Science and Computation*, pages 55–78, 2006.

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