Final

Model Checking for Network Security Requirements via a Flexible Modeling Framework

John D. Powell
Jet Propulsion Laboratory,
California Institute of Technology
John.Powell@jpl.nasa.gov

David P. Gilliam
Jet Propulsion Laboratory,
California Institute of Technology
David.P.Gilliam@jpl.nasa.gov

Abstract

Network security requirements are a complex set of system rules which, if violated can have serious repercussions. Verifying a given system's ability to meet its requirements is of increasing importance as dependency upon networked computer systems grows. Servicing multiple Network Aware Applications (NAAs) results in an operational environment that compounds verification complexity further. While an NAA may be free (or nearly free) from vulnerabilities as a stand-alone software component, it may present serious security risks when interacting with other applications on a network. These vulnerabilities can be exploited with disastrous results.

Verification of network security system properties over concurrent processes (NAAs) is a problem well suited to model checking. However, the inherent complexity in these systems results in an intractable number of possible event combinations. This problem manifests itself as the "State Space Explosion" problem, which is a known limitation of model checkers. This paper proposes an approach that mitigates this problem. The Flexible Modeling Framework (FMF) facilitates construction of a system model in a modular fashion with well-defined methods of interaction between processes. This allows a series of models to be created efficiently. Results from the FMF serve as input to a compositional analysis approach also proposed in this paper to verify properties over models, which would otherwise be beyond the capability of current state of the art model checkers.

1. Introduction

This paper presents a model checking approach that will allow a practitioner to verify critical requirements of a system that must operate in an environment that is currently too complex to be effectively analyzed by state of the art model checkers such as SPIN [1,2,3]. Networked systems, particularly those connected to the Internet, provide a readily available domain of systems meeting these criteria [18].

The Internet is a ubiquitous networking environment that has become a major source of worldwide communication and commerce. Thus, a system that allows Internet connectivity to NAAs must provide security to those applications in a wide range of environments that are continually evolving. Network security requirements seek to prevent or mitigate damage from electronic attacks on the networked system in question. Attacks to interconnected systems can cause either focused or widespread havoc, from system intrusions to spreading of viruses. A major concern is the presence of Distributed Denial of Service (DDoS) attacks that flood victim networks with spurious traffic degrading network communication and preventing access to network resources.

Section 2 provides a brief introduction to model checking. Section 3 discusses the FMF, which is a specific approach to model checking systems that operate in complex and volatile environments. Section 4 will explain the characteristics of networked systems and DoS attacks that motivate the use of this approach in the network security domain. Section 5 relates the FMF to the network security research currently being conducted at the Jet Propulsion Laboratory (JPL). Section 6 will discuss future extensions to the FMF to allow other domains to take advantage of this novel approach.

2. Model Checking

Model checking involves the building of a mathematical model of the system and identification of properties to be verified. Model checkers, such as SPIN [1,2,3], are tools that automate the process of verifying a property over its corresponding model. This is accomplished by an exhaustive search of the *state space* generated by a model. State space refers to the collection of reachable system states represented by the model. A

given state consists of the collection of all variables in the model and their associated values at a given point in time. When the value of a **single** variable changes, a transition to a new state in the state space occurs. [1,2]

Model checkers suffer from the known problem of *state space explosion.* [1,2,3] State space explosion refers to the exponential growth (m^n) in the size of the state space with respect to the number of variables (n) and their ranges (m) in the model. This results in a limitation on the size and or complexity of a system that may be model checked. Many techniques have been developed to advance model checking and to mitigate the state space explosion phenomenon to varying degrees. [2,3,4,5,6,7,8,9,10,11,12, 13,14,15]

Abstraction of system behavior requires the combination of domain and model checking expertise to ensure that behavior not pertinent to the property set is abstracted away in such manner as to minimize the state space while leaving pertinent behaviors intact. However, the system's necessary environmental behaviors often produce much larger state space explosions than the system itself. Currently, reduction of the environment's state space relies on the ability of practitioners to make assumptions about the future environment in which the system will operate. These assumptions represent a weak link in the confidence of the formal verification results in relation to the deployed system. Further, formulation of adequate assumptions may not be possible due to the unpredictable evolution of an environment during operations. A system connected to the Internet produces this very scenario. The end result is a state space that cannot be sufficiently reduced to employ model-checking technology. The FMF seeks to overcome this barrier.

3. Flexible Modeling Framework

The FMF is well suited to verification of a system with respect to a highly complex environment. The basis of the approach is the notion of building the model in components that conform to a standardized interface for interaction between components. This interface is not designed to control interaction (i.e., interleaving of events) but to allow components to be assembled in a uniform manner to easily form multiple models each representing a subset of the full system model.

3.1. Modeling in Components

The practice of building a model in components seeks to build a series of very small models referred to as components. Each model component will represent a single logical segment of the system behavior. Multiple segments will have to be assembled to form a non-trivial model of a system, subsystem or environment behavior. This is done through the framework via a previously agreed upon set of variables (defined interface) which facilitate communication between components when necessary. Finally the model that will be verified with respect to a given property or properties will be *assembled* from the components within the framework.

Each component must be designated as a *Core Component* or *Non-Core Component.* Core Components are those model components that must be included in every assembled model. A component is a core component if:

- It represents system (non-environmental) behavior.
- It **does not** contain modeling constructs that force synchronization or controlled interleaving with respect to another component.

Components that combine system behaviors and environmental behaviors should be diligently avoided. These components can often be split into two or more components that do not combine environmental and system behaviors. However, if this combination in a single component cannot be avoided it must be treated as a core component based on its system behavior portion. Note that there is an increase in state space size for every assembled model when components combining system and environmental behaviors are used unnecessarily. This can place increased limitations on Compositional Verification.

3.2. Compositional Verification

Compositional Verification attempts to produce verification results on assembled models that represent the system (S) and a subset of the environmental behavior (e) in a manner that allows those results to be generalized to the system and its environment at large (E). Any synchronization or controlled interleaving behavior (Syn) that exists in E prohibits generalization of verification results over ($S \cup e$) to ($S \cup E$) unless ($Syn \subset e$). (See Section 3.2)

The basis of the compositional verification approach is the verification of a system (S) with regard to a subset of its environment (e) in a manner that allows those results to be extrapolated to the environment at large (E). First, a model of the system S is built. However, the behavior of this model is only interesting with respect to E. In many cases, the addition of model E to S produces a state space explosion that overwhelms the capabilities of model checkers. The environmental state space explosion problem is seen in model checking efforts across various domains.

The proposed solution of verifying a large model by examining sub-models within it rests on the idea that, with some restrictions, an event sequence that produces a property violation found in a given model (m) will still exist in a larger model (M) where $m \subset M$. Identification of a property violation is the discovery of a sequence of events (transitions though a state space graph), which represents behavior contrary to the property specification. Within certain restrictions (R), if a property violation is found to exist in m, then it also exists

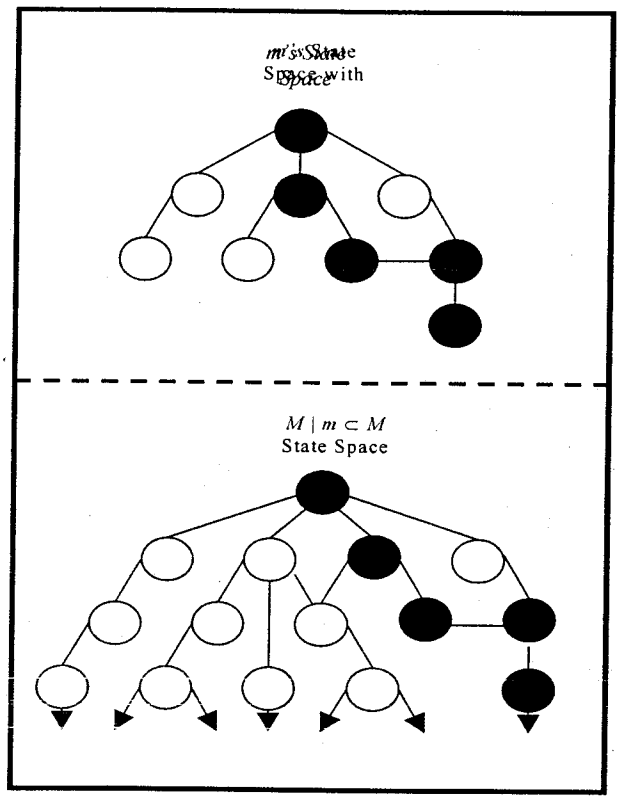

FIGURE 1

in model M such that $m \subset M$. (See Figure 1) While the use of a model checker may be infeasible for models such as M due to state space explosion, the smaller model m presents a feasible alternative. Further if R holds for m then the results may be readily extrapolated to M.

The characteristics of the FMF, when applied to a system M, ensure that all legal component combinations (m) within the framework R will hold. Therefore, valid extrapolation of verification results from m to M is preserved. This will eliminate the need to exhaustively examine all possible m's when a property violation has been found. Thus, the average number of m's subject to model checking a single property is reduced.

Prove that if a path P exists in a graph G then P exists in all graphs G'' where G is a sub-graph of G''.

1. *Let G = (V, E) be a graph (assumption – G exists)*
2. *Let G' = (V', E') be a graph (assumption – G' an arbitrary graph exists)*
3. *Let (G'' = (V'', E'')) = G ∪ G' (assumption – G'' exists)*
4. *Let P = {v1, v2, v3 ... vn} | P ⊂ V and (∀(i, | 1 ≤ i <n), (vi, vi+1) ∈ E) (assumption – P exists in G)*
5. *By 2 and 3, V ⊂ V''*
6. *By 2 and 3, E ⊂ E''*
7. *By 4 and 5 and set theory, P ⊂ V''*
8. *By 4 and 6 and set theory, ∀(i, | 1 ≤ i <n), (vi, vi+1) ∈ E''*
9. *By 3 and 7 and 8, ∀(G'' | G'' = G ∪ G'), P ⊂ V'' and (∀(i, | 1 ≤ i <n), (vi, vi+1) ∈ E'')*
10. *By 9 and 4 if P exist in G then P exists in G''*

3.3. Restrictions

As mentioned above, compositional verification extrapolates results from partial model instances assembled from components within the FMF when the set of possible model instances adheres to certain restrictions. The FMF is an approach that preserves conformance to these restrictions during the model building process while also affording the modeler a reasonable level of flexibility.

3.3.1 Shared Start State, The first restriction is that all model instances must have the same start state. A node within a state space represents all the variable/value pairs in the model at a given point of execution. This "given point" is referred to as a state. When a single variable/value pair is legally altered, a state transition takes place. Graphically, the current search of the state space graph moves to an adjacent node. When a violation is found in a sub-graph of a larger model, extrapolation to the model at large is valid if and only if the start state (node) is the same for both state space graphs. This restriction can be formally stated as:

A path P in graph G that is reachable from node S is reachable in graph G'' if $G \subset G''$ and S is always reachable in G''.

3.3.2. Absence of Synchronization in Omitted Model Components, Synchronization and other blocking behavior between components can affect reachability within the underlying state space. Therefore, if a property violation is found in a sub-model (M), this result cannot be validly extrapolated to the model at large in all cases. When synchronization and blocking behavior has been omitted from M, it is possible that a path P representing a property violation is not reachable in all model instances that include M. The formal statement in section 3.2.1

must be extended to include this restriction. The formal statement becomes:

A path P in graph G that is reachable from node S is reachable in graph G'' if G ⊂ G'' and a path {S, P', P} exits in G'' and S is reachable in G''.

The existence of synchronization or other blocking behaviors in a component C such that ($C \in \sim(G \cap G'') \wedge C \notin G$) introduces the possibility that path P' may exist in G but not in G''. Thus, while the property violation technically exists in the state space, the overall system contains behaviors that prohibit it from ever being reached (i.e., fault protection).

S may be reachable without being the start node in G''. However, the nature of many temporal properties restrict what may or may not take place prior to the start of path P. [16,17] While the assertion that S must be reached by a path {P'', S} in G and G'' and P must be reached by a path {S, P'} in G and G'', this property can be too restrictive. While it guarantees valid extrapolation via identical precursory events, the modeler's flexibility would be severely impeded.

Finally, some systems have synchronization and blocking behaviors that must be included in the model. Within the framework, these synchronization behaviors will be modeled in components either exclusively or in conjunction with other behaviors. These components must be included in every instance of the model that is to be assembled to ensure that the path reached in a model M can be extrapolated over components that were not included. It is important to note that there will always be a core set of components that must be included in every model instance. They include components containing critical system behavior (as opposed to environmental behavior) in addition to synchronization and blocking behaviors. Finally, at least one component containing the start state S must be included. In most cases, at least one system critical component will already contain this behavior. If this is not the case, an "artificial" component containing a safe artificial start node can be inserted.

3.4. Automation

For illustrative purposes the automation process will be explained in terms of the network security example. The potential for automation to maximize the benefit of using the FMF is dependent on one's ability to:

1. Define and conform to a consistent interface definition between components
2. Systematically tailor the interaction of generic network system software (E S y) and environment (E x S) into their minimized counter parts E S i and E i S | i∈ (x∩ y). (See Figure 2)

If condition 2 above can be achieved in an automated fashion, it would be of great benefit also to automate the NAA combination selection and property selection. This would allow models to be built, minimized and prepared for verification on the fly. It would reduce the subsequent tasks for fully automated verification of multiple models/properties to a scripting problem. The overall goal of this approach is to allow a model-checking practitioner to build (M+2) components once and verify the relevant properties for MN variations of the system model automatically (See Figure 2). (Increase the size of some of the boxes where the verbiage is cut off.

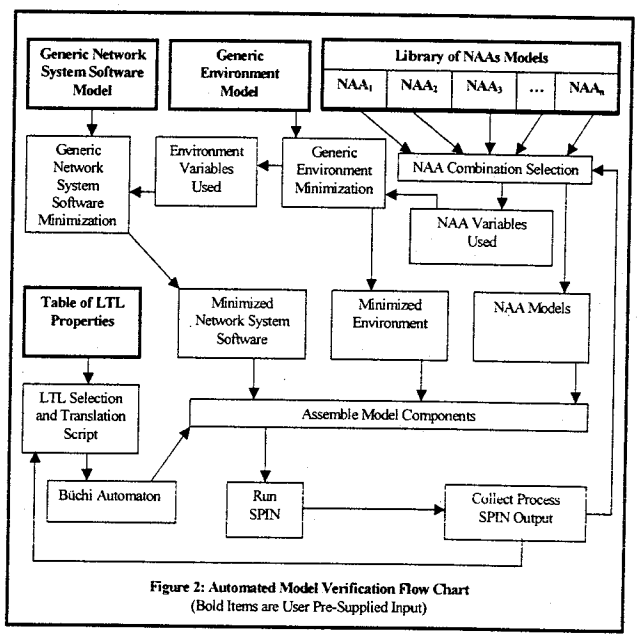

Figure 2: Automated Model Verification Flow Chart
(Bold Items are User Pre-Supplied Input)

The scripting problem involved in completing automation of this process is a combination of file management commands and SPIN command line calls. After the NAA combination has been identified, the minimization tasks are performed. The variables that are in use as a result of the NAA combination are supplied to an environment minimization routine along with the generic environment model. This routine's output is the minimized environment (unused environment variables disabled) and the environment variables in use. The network system software minimization routine accepts the environment variables in use and the generic network system software model as input. Next, a minimized network system software model is output (unused network system software variables disabled). The user pre-supplied table of LTL properties is accessed. An appropriate property is selected and translated into a bücci

automaton via a command line call to SPIN. The following four components are assembled into a model:

- NAA Models
- Minimized Environment Model
- Minimized Network System Software Model
- Bücci Automaton Representation of a Property

Finally the SPIN model checker will be run to verify the property and the output will be collected / processed. The process will iterate by obtaining another property for the current NAA combination (if it exists) or identifying a new NAA combination. The appropriate portion of the process will be repeated depending on the choice made in the last step. This iteration may continue until all properties for all legal combinations of a predefined cardinality are verified.

3.5. Resolution Flexibility

Recall that the size of each component used in a model constructed under the FMF contributes to the overall size of the assembled model by a combinatorial factor (See Figure 3). Also note that these components represent behavior indicative of interaction between an application with the network and other applications (See Figure 2). The abstraction techniques used to produce minimized behavior can hinder property verification when they omit behavioral details that are pertinent to the property in question. The necessary reduction of state space size is inherently in conflict with the need to verify a property over all possible behaviors of the system. Therefore these two competing requirements must be balanced by using techniques including but not limited to:

1. System Abstraction -- Justifying that some system behaviors can in no way affect the property being tested.
2. Domain Specification -- Making some assumption(s) about the behavior that the environment will inflict on the system.

System abstraction and domain specification are non-trivial tasks. First, justification that one behavior is not affecting others is limited by expert's knowledge about number of known system interactions. Further, the number of interactions that must be known and understood may be intractable when overlapping chains of multiple interactions are considered. Therefore, this justification process can mask hazards in software that are caused by the very (yet to be known) process interactions for which model checking is being employed to identify.

Secondly, the domain of possible behaviors in a system's environment can be much larger in scope than the system's realm of possible behaviors. This is true for a networked computing environment. When the environment is too large to model in its entirety, often

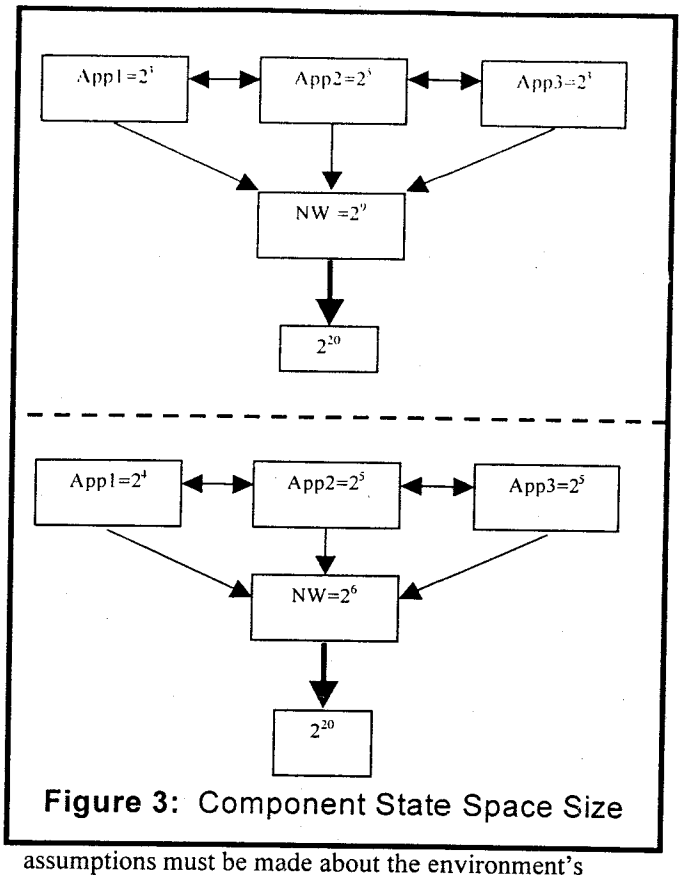

Figure 3: Component State Space Size

assumptions must be made about the environment's possible behaviors. Assumptions often must be made with weaker justification than abstractions made within the system because:

- A system must be fault tolerant across the entire environment domain in which it is deployed
- Experts often have less information about the intricate behaviors of the system's environment at large.
- Environments evolve over time as systems are maintained and adapted to new purposes or software is reused.

The assumptions, if made incorrectly, may produce weaknesses in the analysis results that leave software hazards undiscovered. If the environment changes, altering a model to analyze the new environment may be expensive if the model is not built in such a way that allows for this future flexibility. Achieving flexibility on the model of networked system environments is of particular importance because:

- It is a given that this environment will evolve over time.
- The environment is so complex that the state space remains overwhelmingly large even after reasonable assumptions are made.

This framework offers a tradeoff scenario to building and altering models to verify given properties. By compartmentalizing functional pieces of the model, each component's state space can be calculated and associated with the component. A model's state spaced can be

calculated at or prior to assembly time. When the model checking capability limits are reached, the model may be altered in a very systematic way. Insight into behavior associated with a given property may be achieved by increasing the level of detail with which a component's behavior is modeled. This will increase the assembled state space accordingly. Thus, another component's detail must be decreased to stay within a model checker's capability. The framework allows the modeler to choose competently which component(s) to apply abstraction to and how much abstraction must take place (i.e., how much state space must be removed) to continue to perform model checking (See Figure 3). This abstraction minimizes the number / severity of assumptions that must be made to compensate for added detail in other areas by predetermining the amount of state space that must be eliminated in a given component. By examining the model with varying levels of behavioral resolution in different components, verification results may be achieved via Compositional Model Verification techniques (See 3.2).

4. Network Systems

4.1. Architecture

The architecture of network systems is by necessity designed to provide a ubiquitous means for transferring arbitrary data between computing systems. Thus, these computing systems share and make use of data via the network and their software.

For purposes of formally verifying network system software via model checking, the architecture may be initially viewed as two major layers. *Layer 1* is the NAAs. These are software programs that rely on the network while performing their function but do not implement the network communication functionality. *Layer 2* is the network system software. This is the software that is responsible for providing communication between the NAAs and other systems connected to the network. This includes such functions as send, routing, receiving and returning data packets appropriately over the network.

When analyzing network software for violations of security requirements the network software is the system in question. The combination of all the NAAs interacting with the network at a given point in time is analogous to the environment of the network software system. Due to the large number of NAAs in existence the number of possible environments is far too large to examine each one individually. The problem becomes worse when NAAs must be examined in combination, and are intractable when combinations must allow for multiple instances of one or more NAAs. The representation of an environment that could display any and/or all of the behaviors resulting from possible NAA combinations as described above is extremely complex. The FMF is well suited to verification of a system with respect to a highly complex environment.

4.2. Security Requirements

Network requirements most often take the form of some succinct statement about an interaction between the network system software and its environment. For example, "Every messages sent over the network would either be received or returned to the sender". These rules are defined by the network protocol in use. In the case of the Transmission Control Protocol / Internet Protocol (TCP/IP), standard rules are set and all network system software must obey the rules to communicate. These rules are usually grouped together into sets and follow the OSI network model. In TCP/IP the Internet Control Message Protocol (ICMP) is used, in part, to help network packets reach their intended destination and to provide error reporting when the destination is unreachable, latency or other issues cause packets to get lost. The *Ping* command is an example of network software whose functionality obeys the requirement that every message sent over the network will either be received or returned. The Ping command can be used to verify connectivity between your host and a given Internet destination by simply typing ***ping*** and the destination_host name or IP address. A ping packet is sent from your host to the destination. The ICMP protocol designates a series of replies. If the packet reaches the destination, the destination_host replies with ICMP Echo Reply packets (See Figure 4 below). If the packet cannot find the destination_host, ICMP Error Replies are returned to your host.

Operational requirements specify the behaviors that a networked system should exhibit such as accessibility by authorized users and denial of access by unauthorized users. Security requirements seek to safeguard the operational requirements from malicious or inadvertent environmental events that attempt to violate them. These environmental events, referred to as attacks when perpetrated maliciously, may be categorized as:

- Those that cause network software malfunctions that produce behavior that violates operational requirements.
- Those that exploit "correct" network software behavior in order to force the system to violate operational requirements. Many DoS

attacks fall into this category (See Section 3.3).

1. Pinging 198.49.45.10 with 32 bytes of data:

2a. Reply from 12.126.0.29: Destination net unreachable.
3a. Request timed out.
4a. Reply from 12.126.0.29: Destination net unreachable.
5a. Request timed out.

2b. Ping statistics for 198.49.45.10:
3b. Packets: Sent = 4, Received = 2, Lost = 2 (50% loss),
4b. Approximate round trip times in milli-seconds:
Minimum = 0ms, Maximum = 0ms, Average = 0ms

Figure 4: 2 Possible "Ping" Scenarios

Combating attacks involves *recognition* and *response*. Recognition relies on previous encounters with, and thus previous damage from, the attack in question. In the first category, damage from subsequent attacks of the same type may be minimized or avoided though effective response. The latter category, which is the primary focus of this ongoing research, presents a much harder response problem because the early event in the attack are 'correct" network software behavior and thus currently cannot be distinguished from normal network events.

4.3. Attacks

Denial of Service (DoS) attacks, the most recent Internet plague, is having dramatic effects on network service and stability of its victims. The current state of the art is that of "react and patch" the network software for each new attack encountered. Often with a DDoS attack, recovering may take hours (or even longer).

Although DoS attacks are not something new, the increased accessibility of the Internet and the ever decreasing age and sophistication of the average computer hacker coupled with the complexity of hacker code proliferated across the Internet is resulting in an enormous surge in the type of attack which is specifically and solely intended to deny service to a given system or application. In many cases, the exploit code to conduct these attacks are freely available on the Internet, and it can affect the stability of the system only by a few keystrokes and by a mere click of the mouse.

These attacks take advantage of the deficiencies in the TCP/IP protocol, which is used as the baseline for communications on the Internet, and they are difficult, if not impossible, to trace their source since the packets can be "spoofed" or "forged" as they come from any source on the Internet. DoS attacks attempt to deny network services to authorized users by overwhelming networks and systems via activity generated from multiple sources. This denial of service stems from two causes. First, deadlock will cause a set of processes to cease providing service. The second cause is starvation, where a number of processes vie for resources and some processes never obtain them.

DoS attacks are growing in number and sophistication. The increased complexity of this type of environmental attack makes the need for formal verification of network security requirements of paramount importance as reliance on networked systems grows.

5. Modeling a Networked System in the Flexible Modeling Framework

The FMF Approach has been developed in response to an effort at the Jet Propulsion Laboratory, California Institute of Technology to formally verify vulnerabilities in networked systems as a result of weaknesses in network system software and/or Network Aware Applications (NAAs) interacting with one another and the network in a concurrent fashion.

The application of the FMF to this problem entails the modular development of model components representing network system software behavior. These components form the core components in the FMF. A taxonomy of known NAAs interactions with a networking system is being developed. The non-core components that represent environmental behavior will be modeled from atomic entries in the taxonomy and behaviors indicative of DoS attacks. The properties to be verified will express the result of successful DoS attacks. Finally, the automation process described in section 3.4 will be employed to produce verification results on sub-models assembled from components in the FMF that may be readily extrapolated to the model resulting from the combinations of all the components developed. Thus, effectively extractive, formally valid, model checking results from an overall system model that to date is too large/complex for state of the art model checkers to verify directly.

6. Extendibility to General Requirements Engineering

Many complex systems are not subjected to formal verification because of cost and complexity issues that make this much-needed form of assurance impractical. The extensibility of this approach to the widest domain of

problems offers a means of formal verification to projects that were previously unable to take advantage of this level of software assurance. This first natural step is to extend the work currently being done to various network protocols and environments. Subsequently, a generic extension to systems in general that must operate in complex environments is feasible

7. Conclusion

A system that allows Internet connectivity to NAAs must partially bear the burden of security at the network system software layer and fully at the NAA level in a volatile evolving environment. The goal of preventing damage from DoS attacks and mitigating damage when prevention is not possible has been an ongoing effort that can benefit from the use of formal approaches such as model checking. Attacks from a network's environment can cause widespread damage. The growing dependence on networked computer systems raises the criticality and severity of the risk that a weakness may allow successful DoS attacks. When a victim network is flooded with spurious traffic preventing network communication and access to network resources by its authorized users, the damage can range from temporary annoyance to severe safety risks and financial losses (as evidenced by those attacks against Microsoft, Yahoo, NASA, an many others).

The model checking approach described in this paper will allow a wider usage of formal verification via model checking by projects where complexity issues previously prohibited it. The novel approach to addressing the environmental complexity and associated state space explosion give the practitioner a means to cope with this phenomenon while retaining the power of formal methods. The current focus on networked systems connected to the Internet provides a sufficiently complex domain of systems to prove the effectiveness of this approach.

8. Acknowledgement

The research described in the paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration

9. References

[1] G. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, Upper Saddle River, NJ, November 1990

[2] G Holzmann, "The SPIN Model Checker", *IEEE Transaction on Software Engineering*, 23:279-295, 1997

[3] J. Powell, A Graph Theoretic Approach to Assessing Tradeoffs on Memory Usage in Model Checking, *Thesis*, West Virginia University

[4] H. Andersen, "Partial Model Checking", *Proceedings 10th Annual IEEE Symposium on Logic in Computer Science*, 1995, LICS95

[5] J. Baumgartner and T Heyman, "An Overview and Application of Model Reduction Techniques in Formal Verification", *IEEE International Performance Computing and Communication Conference IPCCC,*, 1998, pp. 172-177

[6] J. Burch, E Clarke and K. McMillan, "Symbolic Model Checking", *IEEE Symposium on Logic in Computer Science (LICS90)*, 1990, pp.428-439

[7] G. Holzmann and A. Puri, "A Minimized Automaton Representation of Reachable States, *Software Tools for Technology*, Springer Verlag, 1990

[8] E. Clarke, D. Long and K. McMillian, "Compositional Model Checking", *Proceedings of the 4th Annual Symposium on Logic in Computer Science*, 1998, pp 353-362

[9] A. Pardo and G. Hatchtel. "Incremental CTL using Model Checking using BDD Subsetting", Proceedings of the Design Automation Conference, 1998, pp. 457-462.

[10] E. Clarke and T Filkorn, "Exploiting Symmetry in Temporal Logic Model Checking", *5th International Conference on Computer Aided Verification*, 1993

[11] F. Schnieder, S. Eaterbrook, J. Callahan, G. Holzmann, W Rienholtz, A. Ko and M Shahabubbin, "Validating Requirements for Fault Tolerant Systems using Model Checking", *IEEE International Conference on Requirements Engineering*, 1998

[12] C. Meinel and C Stangier, "Increasing Efficiency of Symbolic Model Checking by Accelerated Dynamic Variable Reordering", *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 1999, pp.760-767

[13] W. Chan, R. Anderson, P. Beam, S. Burns and F. Modugno, "Model Checking Large Software Specifications", *IEEE Transaction on Software Engineering*, Vol 24 (July 7), 1998 498-520

[14] M. Dwyer and C. Pasareanu, "Filter-based Model Checking of Partial Systems", *Proceedings of the ACM SIGSOFT Sixth International Symposium on the Foundation of Software Engineering*, November 1998

[15] R. Bharadwaj and C Heitmeyer, "Model Checking Complete Requirements Specification Using Abstraction", *Automated Software Engineering*, 1999, pp. 37-68.

[16] M. Dwyer, G. Avrunin, J. Corbett, "Patterns in Property Specifications for Finite-State Verification", *Proceeding of the 2nd Workshop on Formal Methods in Software Practice*, 1998, pp7-17

[17] M. Dwyer, G. Avrunin and J. Corbett, "A System Specification of Patterns", *http://www.cis.ksu.edu/santos/spec-patterns/*, 1997

[18] D. Gilliam, J.Kelly and M. Bishop, "Reducing Software Security Risk Through an Integrated Approach" Proceedings of IEEE 9th International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, June 2000

Due to NASA's increasing reliance on software there is a growing need for rigorous formal verification of software. Formal Methods (FM) is a collection of techniques for formally specifying a system and verifying its requirements. FM has been successfully used in the hardware arena but its application to software is relatively immature. FM offers powerful techniques to detect hard-to-find flaws and inconsistencies early in the software development lifecycle.

The current barriers to usability for FM include complexity issues and cost. Thus FM remains a somewhat disjoint set of techniques that are expensive to apply. Domain and FM experts must choose small portions of a system for FM analysis due to cost and complexity issues. However, the nature of some FM techniques lends themselves to the possibility of extrapolation. Results from a chosen portion of the system may be validly extrapolated to the system at large if the given FM specification is developed with this in mind. The discovery and development of the methodologies to preserve this validity is a necessary step in increasing FM capabilities. Developing systematic processes for the effective use of FM can alleviate the high cost of FM. The FM novice can be guided to build specifications that offer the benefit of extrapolation of results by using well-defined systematic processes and minimal training. The reduction in FM expertise needed to follow such processes will significantly decrease the expense of applying FM to a given system. As with any well-defined systematic approach the opportunity for (semi-) automation exists which will further lower the overhead of FM usage. In addition to cost benefits, NASA software systems will benefit from increased rigor in software assurance efforts because:

1. A larger portion of a software system subjected rigorous formal verification.
2. When a larger portion of the system is formally verified, the opportunity to prove system-wide validity of FM results through extrapolation is increased.
3. System's currently too large and/or complex for FM techniques will be able to exploit FM technology when using the methodologies and techniques to be developed in this research.