

**Reducing Software Security Risk through an Integrated  
Approach Research Initiative**

**Model Based Verification of the  
Secure Socket Layer (SSL) Protocol**

**John D. Powell**

<b>1. Introduction</b> .....	1
1.1. The RSSR Research Project.....	1
1.2. MBV using Model Checking and the FMF .....	2
<b>2. The FMF Verification Methodology and Rationale</b> .....	4
2.1. Propagation of Verification Results.....	6
2.1.1. Verification Values .....	6
2.1.2. Confidence Ratings.....	7
2.2. Integrating FMF with other RSSR Instruments .....	8
<b>3. Modeling the SSL Communication Protocol</b> .....	9
<b>4. Results of SSL Communication Protocol Verification</b> .....	21
<b>5. Conclusion</b> .....	22
<b>References</b> .....	24
<b>Appendix A: Promela Code Excerpts</b> .....	25
<b>Appendix B: SPIN Output for Verification Results</b> .....	37

# 1. Introduction

This document discusses the verification of the Secure Socket Layer (SSL) communication protocol as a demonstration of the Model Based Verification (MBV) portion of the verification instrument set being developed under the Reducing Software Security Risk (RSSR) Trough an Integrated Approach research initiative. Code Q of the National Aeronautics and Space Administration (NASA) funds this project. The NASA Goddard Independent Verification and Validation (IV&V) facility manages this research program at the NASA agency level and the Assurance Technology Program Office (ATPO) manages the research locally at the Jet Propulsion Laboratory (California institute of Technology) where the research is being carried out.

## 1.1. *The RSSR Research Project*

The National Aeronautics and Space Administration (NASA) has tens of thousands of networked computer systems and applications. Software Security is a major concern due to the risk to both controlled and non-controlled systems from potential lost or corrupted data, theft of information, and unavailability of systems, especially mission critical systems. The cost to NASA if mission critical systems were compromised, would be enormous if these systems were brought down or erroneous data sent to a spacecraft. The RSSR research examines formal verification of information technology (IT) security of network aware software and systems through the creation of a security assessment instrument for the software development and maintenance life cycle. [1,2,3,4,5,6] The network security assessment instrument is composed of 5 parts:

1. A Vulnerability Matrix
2. Additional Security Assessment Tools (SATs)
3. A Property Based Testing (PBT) Instrument, and
4. A Model Based Verification (MBV) Instrument incorporating a Flexible Modeling Framework (FMF)
5. A Software Security Checklist (SSC)

The vulnerability matrix is part of the UC Davis DOVES database containing vulnerability descriptions and the code used to exploit them. This information is used to extract properties and requirements that express potential network vulnerabilities. The PBT tool and the FMF can then utilize these properties. The SATs are a collection of tools available on the Internet that can be used to test for potential weaknesses of software code. This list includes a description of each of the tools and their uses. It will be updated as additional tools become available. The PBT tool performs formal verification of properties, including those obtained from the vulnerability matrix, at the code level. Properties are verified by slicing the code in search of the specific vulnerability properties in question. Like the PBT tool, the FMF formally verifies properties over the system. However, the FMF performs this action early in the software lifecycle at the abstract level when code may or may not yet exist. The SSC will provide software code developers with an instrument for writing secure code for network aware applications, such as the use of network ports, protocols, authentication, privileges, etc...

It will also ensure that released software does not provide backdoors into or information about an organization's systems or networks.

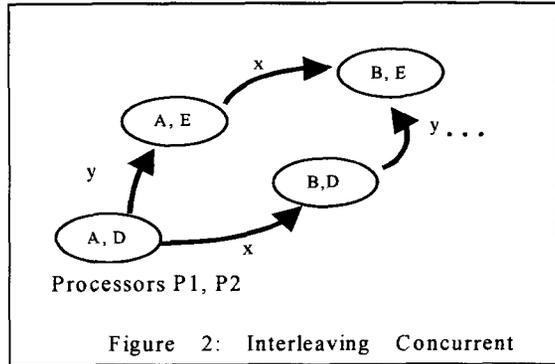
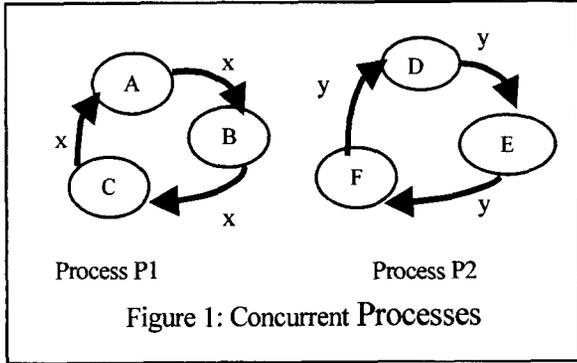
## **1.2. MBV using Model Checking and the FMF**

Model Based Verification makes use of discrete finite models to verify compliance of the modeled system to desired properties. While the FMF is a generic approach to modeling, the specific properties addressed in this document focus on software and network security properties pertaining to the SSL communication protocol. Network security properties often focus on characteristics that are manifested through the operation of multiple software components and systems operating concurrently with or without an attacking process. The concurrent nature of the systems results in an operational space that is too large to verify system properties effectively through traditional testing of the implementation. Further, vulnerabilities introduced in the early phases of the development lifecycle are costly to remove in later phases when an implementation is being tested. This results in the addition of cumbersome workarounds and "patches" to repair the software system which themselves could introduce new vulnerabilities. MBV offers the opportunity to perform verification of properties early in the life cycle, providing a clearer understanding of the vulnerability issues within the system during the design phase before an implementation exists.

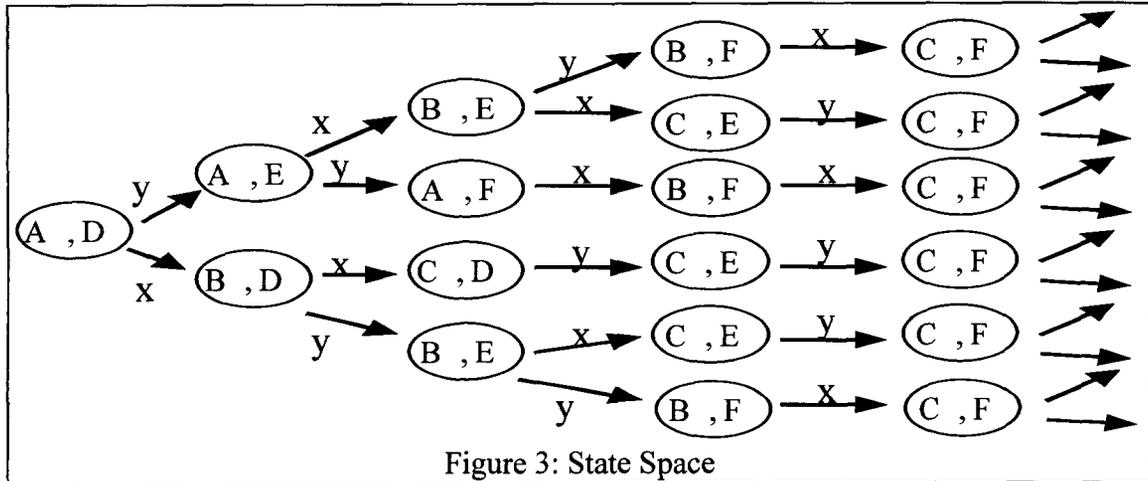
The MBV technique demonstrated and reported on in this document involves Model Checking (MC). Model checkers automatically explore all paths in a finite state space from a given start state in a computational tree. The objective is to verify system properties over all possible scenarios within a model. Model checkers differ from more traditional heavyweight formal techniques in that model checkers are operational, as opposed to deductive. Deductive approaches, while offering a higher level of completeness and more resilience in the face of larger systems, are difficult to apply and require a great deal of expertise.

Model checkers provide counter examples when properties are violated. The counter examples may be used to determine the cause of the property violation and used as representative traces for test case generation. [8,9] Their goal is oriented toward finding errors as opposed to proving correctness since the model is an abstraction of the actual system. Where errors are found in the early lifecycle, sample test specifications can be preserved for use by the PBT to provide traceability verification.

MBV techniques, such as MC, are not without drawbacks. Among them is the inability to model a system with a high degree of fidelity in a timely manner while the system evolves. This is particularly problematic in the earliest stage of development such as concept, requirements and high-level design when the system definition is most volatile. MC's lack of agility limits an analyst's ability to maintain an up-to-date model and minimize the latency between the introduction of errors and their discovery.



A limitation specific to MC is the state space explosion problem. [10] Similar to the growth of the operational space mentioned above, the state space that a model checker must search to verify properties grows at an exponential rate as the model becomes more detailed. As shown in figures 1 through 3 the state space grows at a rate of  $m^n$  where  $m$  is the range of possible values a variable may assume and  $n$  is the number of variables in the model. Despite the use of modeling techniques such as abstraction and homomorphic reduction, it is infeasible to verify many software systems in their entirety though model checking beyond those that are either complex and very small or moderate in size and very simplistic.



An innovative verification approach that employs MC as its core technology is offered as a means to bring software security issues under formal control early in the life cycle while mitigating the drawbacks discussed above. The FMF seeks to address the problem formal verification of a larger system by a divide and conquer approach. First, verifying a property over portions of a software system, then incrementally inferring the results over larger subsets of the entire system. As such the FMF is a:

- System for building models in a component based manner to cope with system evolution in a timely manner.

- Compositional verification approach to delay the effects of state space explosion and allow property verification results to be examined with respect to larger, complex models.

Modeling in a component-based manner involves the building of a series of small models, which will later be strategically combined for system verification purposes. This correlates the modeling function with modern software engineering and architecture practices whereby a system is divided into major parts, and subsequently into smaller detailed parts, and then integrated to build up a software system. An initial series of simple components can be built when few operational specifics are known about the system. However, these components can be combined and verified for consistency with properties of interest such as software security properties.

The compositional verification approach used in the FMF seeks to verify properties over individual model components and then over strategic combinations of them. The goals of this approach are to:

- Infer verification results over systems that are otherwise too large and complex for model checking from results of strategic subsets (combinations) while minimizing false reports of defects.
- Retain verification results from individual components and component combinations to increase the efficiency of subsequent verification attempts in light of modifications to a component.

## **2. The FMF Verification Methodology and Rationale**

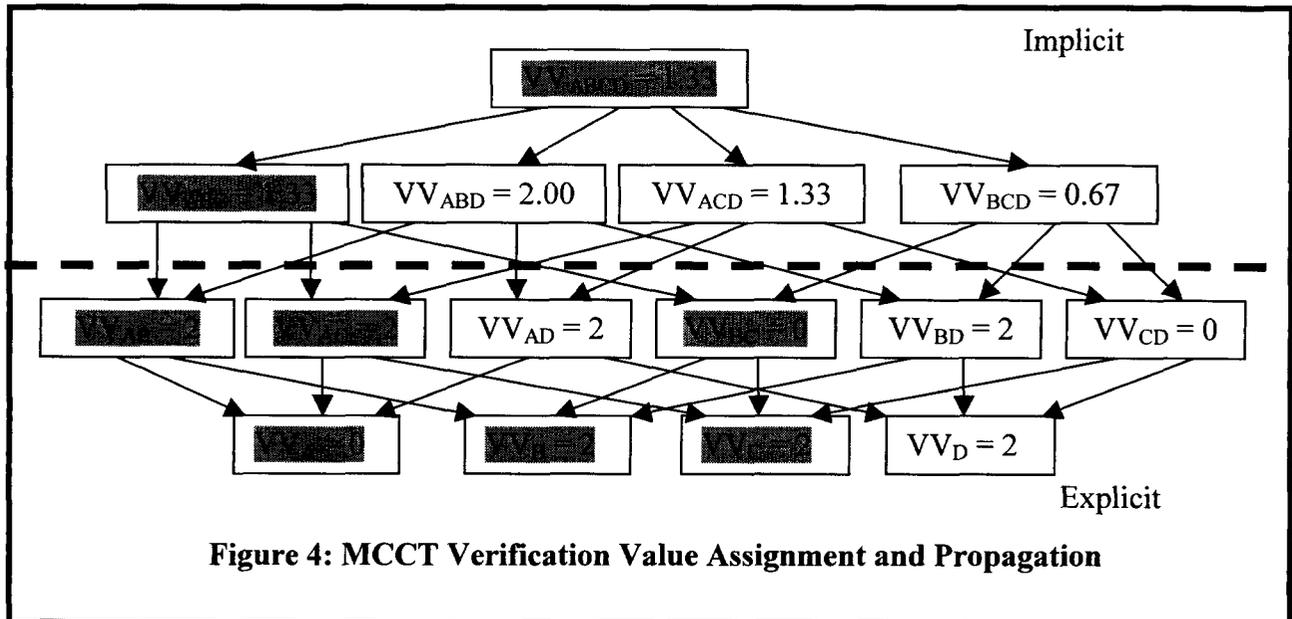
The FMF approach's verification strategy for systems, whose entire model is too large for MC, is first to verify the property in question ( $p$ ) over each individual component individually. Next,  $p$  is verified over unique component combinations of 2 or more components that are built up until no unique component combinations, whose state space may be model checked remain. During this combination and verification process, the relationships between the combinations are preserved such that for two arbitrary combinations  $x$  and  $y$  where  $x$  is a subset of  $y$  and the cardinality of  $x$  equals the cardinality of  $y$  minus 1,  $x$  will be considered the child of  $y$  in a tree structure of model checked model combinations. (See Figure 3) The result of maintaining these relationships is the generation of a tree of verified model component combinations (MCCs) with multiple root nodes. This tree is referred to as the Model Component Combination Tree (MCCT) The tree's leaf nodes consist of single verified components, the parents of leaf nodes consist of a combination of 2 components and their parents consist of a combination of 3 components etc... (See Figure 4)

At some threshold, determined by the amount of available memory present on the verification-computing platform, the state space of MCCs becomes too large for state of the art MC. The combinations beyond this threshold may be systematically computed but not model checked and is referred to as the implicit portion of the MCCT. The portion of the MCCs that can be verified via MC is referred to as the explicit or verified portion of the MCCT.

A root node in the verified portion of the MCCT is implicitly connected to parents that represent MCCs whose state space size prohibits direct model checking due to memory constraints. The only exception to this assumption is when the state space of the entire system model is small enough to be model checked on the platform in a traditional way. However, the FMF approach offers benefits to modeling beyond MC state space explosion and can handle verification of this case as a trivial case where there will exist only one root node which represents the entire system and produces the traditional MC result over the entire system.

The implicit parent of an explicit root node follows the same relationship rules that explicit nodes follow with regard to makeup and cardinality. (See this section above) An explicit “root” node is a subset of its implicit parent and contains exactly one less model component in its combination as illustrated by the shaded boxes in Figure 4. Additionally each parent (implicit and explicit) has a number of children equal to the number of components in its combination. Thus, the entire implicit portion of the MCCT can be systematically generated and probabilistic verification statements made.

In addition to supporting the FMF’s compositional verification approach the MCCT allows for partial re-verification in light of system model changes. As the system is evolves, related model components are updated to reflect changes. The modular style of modeling in components results in a localization of the effects of updating a model component. Since the relationships between components and MCCs and past verification results of model components and MCCs are maintained, only the modified component and MCCs in which it participates need be re-verified. This represents a significant savings in terms of computational efficiency during subsequent (re-)verification executions.



## **2.1. Propagation of Verification Results**

Each node (MCC) in the MCCT is a vehicle for retention of knowledge pertaining to the verification of the MCC represented therein. Each node carries two primary pieces of knowledge directly pertaining to verification.

- A verification value ranging from 0 to 2 that describes whether the property in question holds over the associated MCC. (See Figure 4)
- A confidence rating ranges from 0 to 1 that describes the probability that the verification value produced is correct.

A verification value of 0 indicates that the property decidedly does not hold. In terms of network security properties for software, a property violation represents the discovery of a network security vulnerability. The indication as the verification value progresses from 0 towards 1 is that the predictability of the property not holding is diminishing towards undecidability and is completely undecidable at a value of one. As the verification value moves from 1 to 2 the predictability that the property holds is increasing with a maximal predisposition that the property holds over the MCC when the verification value is 2. The verification value will always be 0 or 2 in the explicit or verified portion of the MCCT because the MCCs are model checkable and thus a definitive verification result is obtained for that MCC. As verification values are derived for MCCs or nodes in the implicit portion of the MCCT the degree to which a property is believed to hold /not-hold over a given MCC begins to vary between 0 and 2 because no direct MC results are available. It bears noting here that the verification value in no way expresses confidence in the verification result expressed by the verification value. Thus, a Verification Value of 1 means that information across the various model components in a MCC is in conflict. A separate confidence rating is used to express the degree of confidence that the heuristics have correctly identified conflicts. (See Table 1 in Section 2.1.2) Verification Values in the implicit portion of the MCCT are calculated by averaging the Verification values of its children. (See Figure 4) In the explicit portion the MCCT, the verification values of a node is not heuristically derived from its children because the application of MC to the parent's MCC is a definitive verification answer. As can be seen in Figure 4 it is possible to systematically propagate verification values over the full set of model components and thus the entire system model.

### **2.1.1. Verification Values**

Verification values assist in determining early lifecycle repairs and assurances for a system before software security vulnerabilities propagate and become exponentially more expensive to repair. In relation to the MCCT these activities are guided by trying to maximize and/or preserve Verification Values across the MCCT. For example, the small system depicted in Figure 4 indicates that model component A ( $VV_A$ ) violates the property but the violation is mitigated individually by both model components B ( $VV_B$ ) and C ( $VV_C$ ). However, B ( $VV_B$ ) and C ( $VV_C$ ) taken together without benefit of any other model component violate the property. Further, even though together they violate the property in question in combination, individually each model component (B and C) satisfies it as well as correcting the property's violation in A ( $VV_A$ ). Consider that this set

of component represents the system at the early design or architecture stage of the development life cycle. Several important questions can be prioritized and addressed. First, what interaction between the behaviors represented by model components B and C causes the property to be violated? Identification of this anomaly guides the network security profession or software developer toward the root cause of the vulnerability. Secondly, does components B and/or C mitigate the property violation found in component A because they are explicitly required to do so or is it coincidental? The documentation must either reflect this reliance on B and C in the form of new / existing requirements or action must be taken to isolate and repair the transient vulnerability.

Similar steps may be taken to address the verification value of 0 in MCC  $VV_{CD}$ . It becomes apparent from this very small example that verification values in the FMF approach generates numerous interrelated questions. However, this is considered a strength of the FMF approach because the questions and issues are brought out very early in the lifecycle for consideration. Further, as early decisions are made and captured, efficient localized updates of the system model are supported through the modular nature of the model components in a very agile manner. Issues affected by subsequent changes are automatically revisited though the required localized re-verification of affected combinations. This last feature of the approach is considered a failsafe and not a substitute for good practices such as documentation of decisions and emergent requirements.

<b>Conf. Rating</b>	<b>Verif. Value</b>	<b>Description</b>
1.0	2.0	Highest confidence of No Property Violation
0.99-0.7	2.0	Reasonably High Confidence of No Property Violation
0.69-0.00	2.0	Questionable to no confidence of No Property Violation
1.0	0.0	Highest confidence of Property Violation
0.99-0.7	0.0	Reasonably high confidence of Property Violation
0.69-0.00	0.0	Questionable to no confidence of Property Violation
0.99-0.7	1.0	Reasonably High confidence Property (Non-) Violation cannot be Predicted
0.69-0.0	1.0	Questionable to Low confidence Property (Non-) Violation cannot be Predicted. Non-Uniform Component Resolution may produce productive predictions

**Table 1: Verification Value / Confidence Rating Explanations**

### **2.1.2. Confidence Ratings**

Confidence ratings for MCCs reflect the degree to which the heuristics of the FMF approach believe that the corresponding verification value correctly decides the result of the property verification over the MCC. It bears noting here that the confidence rating does not serve to improve decideability. Rather it serves to project the confidence in the decision once it is made. For example, when the verification value is at or very near 1 a high confidence rating means that the approach is very sure that the verification answer cannot be derived from the information available. Conversely, a low confidence rating

when the verification value is (near) 1 means that insufficient information to decide the verification is available to decide the verification answer but existing information conflicts to a large degree. Reasoning about confidence ratings over verification values provides useful distinctions that may be used to guide future action. (See Table 1) such as identifying of a portion of the system that is likely to be problematic or under specified.

## 2.2. Integrating FMF with the other RSSR Instruments

The individual parts of the Security Assessment Instrument can be used separately or in combination (See Figure 5). When used in combination they provide the following additional benefits of:

- Reduced rework to identify security properties.
- Increased confidence in the system through verification at multiple times during the development and maintenance lifecycle.
- Use of one tool to verify the input and output of other tools in the network security instrument.
- Finding additional network security attacks yet to be seen in the wild (attacks that have not yet been seen outside of a laboratory environment) and test for their viability and severity.

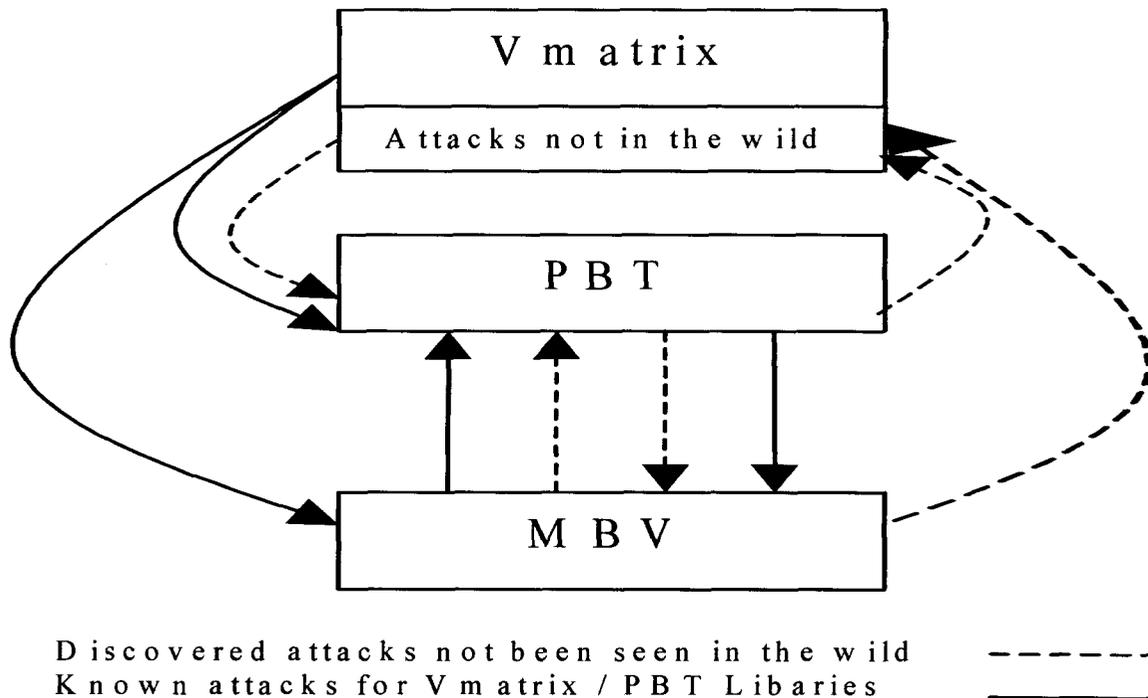


Figure 5

In the network security arena an integrated approach, which includes the FMF as a model-based verification element, for assessing security vulnerabilities is graphically represented by the diagram in Figure 5. The other parts of the Security Assessment Instrument are PBT and the VMatrix. PBT is an approach that allows the analyst to systematically test an implementation for adherence to various properties by making use of a support tool called the Tester's Assistant (TA). [1,2,5,6] First, the property is expressed in a form that the TA accepts as input. Then, an analyst uses the PBT to systematically insert assertions that are pertinent to the property into an implementation and exercises the implementation. The objective is to discover traces through the implementation that produce a non-conforming scenario. The Vmatrix, examines vulnerabilities, exposures and the methods used to exploit them. Vulnerabilities and exposures are listed along with their Common Vulnerabilities and Exposures (CVE) listing. [2] The VMatrix includes:

- A brief summary and a description of the vulnerability or exposure.
- The affected software or operating system.
- The means necessary to detect the vulnerability or exposure and the fix or method for protecting against the exploit.
- Catalogue information, keywords, and other related information as available, regarding the vulnerability or exposure.

In addition to developing an abstract model of a system and performing MC verification, properties of interest must be defined. The identification of specific system critical properties that warrant formal verification is a non-trivial task. Integration of the FMF with the VMatrix addresses this problem for the network security arena. The VMatrix [1,2] provides a searchable knowledgebase from which properties may be extrapolated for use with the FMF in the role of a MC function within the instrument. This is represented in figure 5 by the arrow originating from the VMatrix and ending at the MBV element of the instrument. This step in the integrated approach involves a translation of the property from the VMatrix format to linear temporal logic properties.

This knowledgebase also accommodates the discovery and recording of new network security attacks not yet seen in the wild and that may be discovered through MC techniques. The network security instrument also provides a Property based testing tool [1,2,3] that verifies properties against the actual implementation of a software system. These properties are also extracted from the VMatrix. Used with the FMF, PBT can provide verification of a system implementation's fidelity to the model(s) of early lifecycle artifacts (Requirements and Designs).

### **3. Modeling the SSL Communication Protocol**

The modeling of the portion of the SSL communication protocol of interest for this study did not exhaust memory constraints on the test platform. Therefore, heuristic propagation of results from the explicit portion of the MCCT to the implicit portion was not necessary. That is to say that the current portion of the system functionality that has been modeled resides wholly in the explicit portion of the MCCT because it can be model checked as a whole. However, the component based modeling approach is still useful in

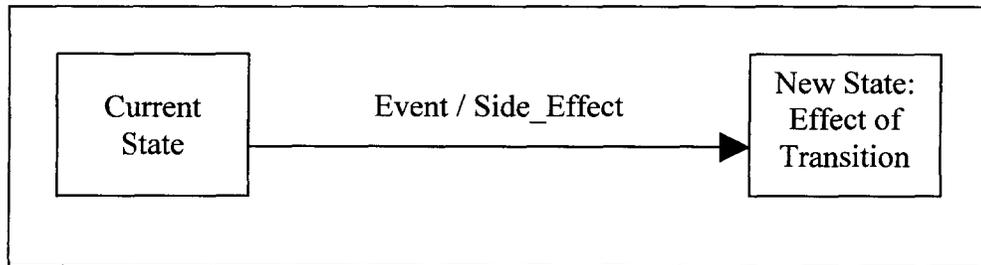
that it can demonstrate which combinations of features within the SSL communication protocol cope with attacks and which combinations are defeated. Further, the component architecture for the verification effort allows for easier extension of the model to study new attacks and additional SSL functionality and varying combinations thereof.

The components are expressed in this document in the form of state charts that are subsequently modeled in the Promela modeling language for use with the SPIN model checker. The state charts are displayed within the text as each component and component combination is discussed. The Promela code excerpts that correspond to the state charts are located in Appendix A and are indexed as follows:

**State Chart *n* from text corresponds to Promela listings *n.1, n.2, n.3*  
... in Appendix A**

The notation used for transitions within the state charts in this document is read as follows (Also See Figure 6)

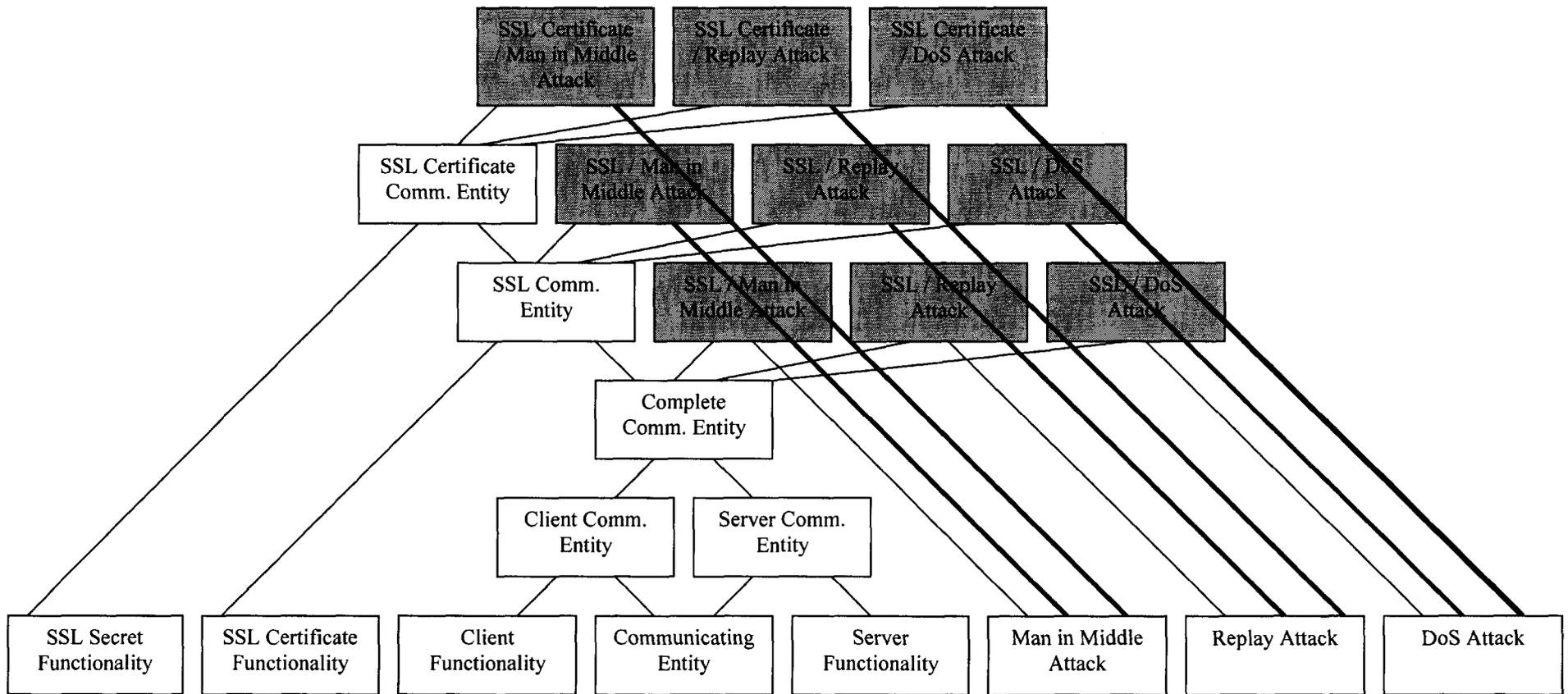
- The current state represents the state of the system before a transition is executed
- The effect of reaching a new state is considered to occur when the new state is reached, i.e. after any side effects of the transition
- Labels on the transition line are of the form X/Y where X is an event that must occur for the transition to execute and Y is a side effect of executing the transition apart from the new state definition. Note that X and/or Y may be null. A null event means the transition may be taken at any time provided the software system is in the “Current State”



**Figure 6: Example State Transition**

Before describing each major component in detail, it is useful to view the overall MCCT structure of the components as they are built up. Some leaf nodes will be shown to be trivial with regard to property verification. Further, it will become obvious that some MCCs will be illogical. For example, a combination that contains attacks but no communicating entities on which to perpetrate the attack is an illogical combination. These illogical combinations are pruned from the MCCT. The trivial leaf nodes are left in place to provide foundation for the more complex elements formed by their combination. The MCCT of the functionality model in this study is illustrated in Figure 7.

Essentially Figure 7 shows how components can be mixed and matched to verify a correctness property over multiple variations of a system's behavior without building a model for each variation. While a model could be built that encompasses all possible behaviors simultaneously, it can be counter productive to effective system analysis. First, combining behaviors in a concurrent manner that do not reasonably co-exist in a system will produce an overwhelming number of false positives. This will flood the analyst with so much data to review that the timeliness of actual system violation will be heavily compromised. Next, upon finding a valid violation of a system's correctness property / properties amid the false positives, the counterexample will often be so convoluted by irrelevant interim model transitions that isolating and recommending corrective action becomes a long and tedious task of analysis in and of itself. When the model is separated into variations through the use of MCCs from the MCCT critical knowledge can be extracted from the pattern of (non-) violations over the model variations as will be seen in the verification results section of this research. This approach often provides a basis for determining critical functionality with regard to property violation and thus isolation of problem areas for corrective actions. Finally, in a open system, such as models of software security protocols and environments, an all encompassing model will unduly stress the limits of the test platform's memory constraints due to excessive state space explosion.

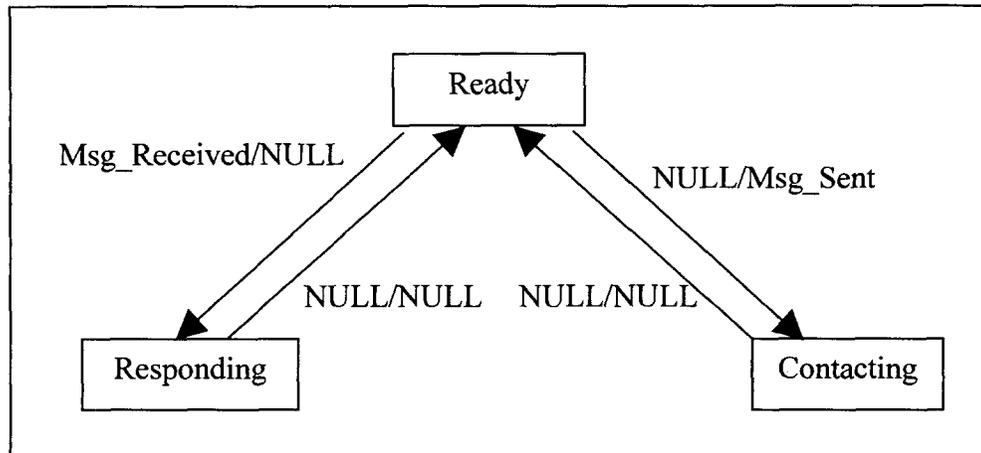


**Figure 7: MCCT for SSL Communication Protocol Verification**

After trivial and illogical MCCs have been pruned for illustrative purposes

Gray boxes represent MCCs that apply to the correctness properties defined in this study

The first software component to be modeled is simple communication without an guarantee of delivery (See Figure 8). Multiple copies of this component are run together in a model component combination along with a component that provides the environmental function of transporting message. A simple event queue was used as the environmental component in the model component combination



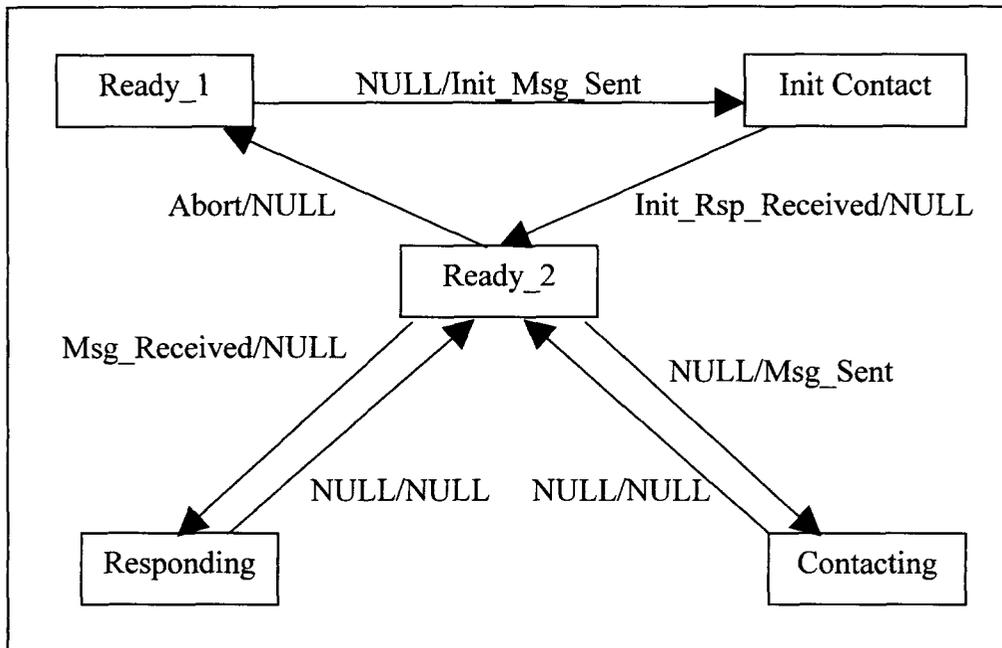
**Figure 8: Simple Communication Entity**

Next, the model of a communicating component is further defined in two different ways. This produced two different components - a communicating client (See Figure 9) and a communicating server (See Figure 10), which have no security characteristics. The main differentiation is that:

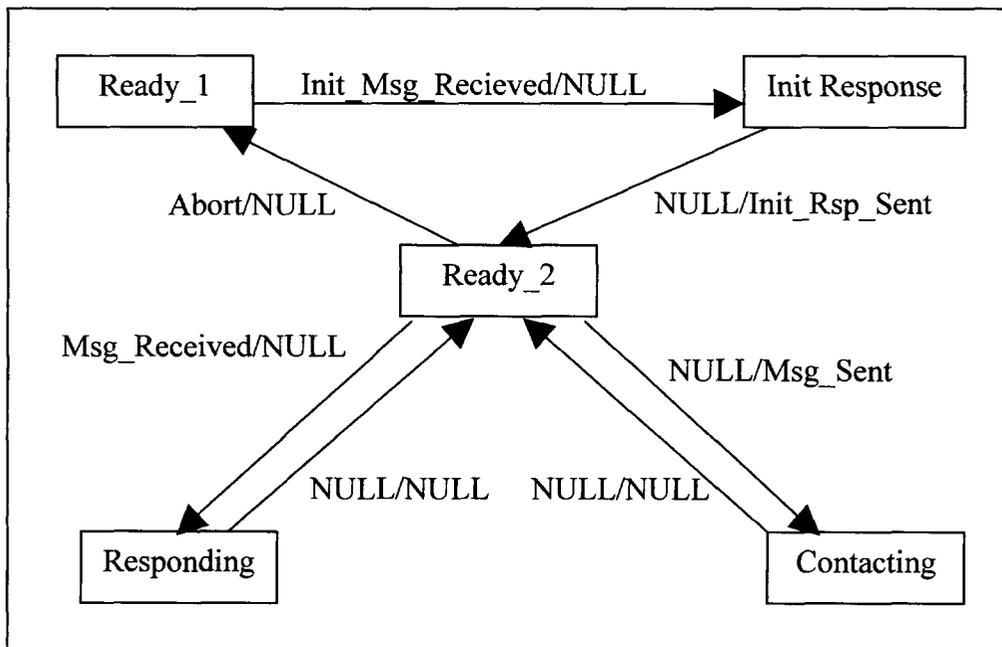
- A client component initiates communication with a server component
- The server component responds to the incoming communication but cannot initiate communication.

After initial contact is made and an initial response is returned, communication between the two components proceed as a simple communication illustrated in Figure 8.

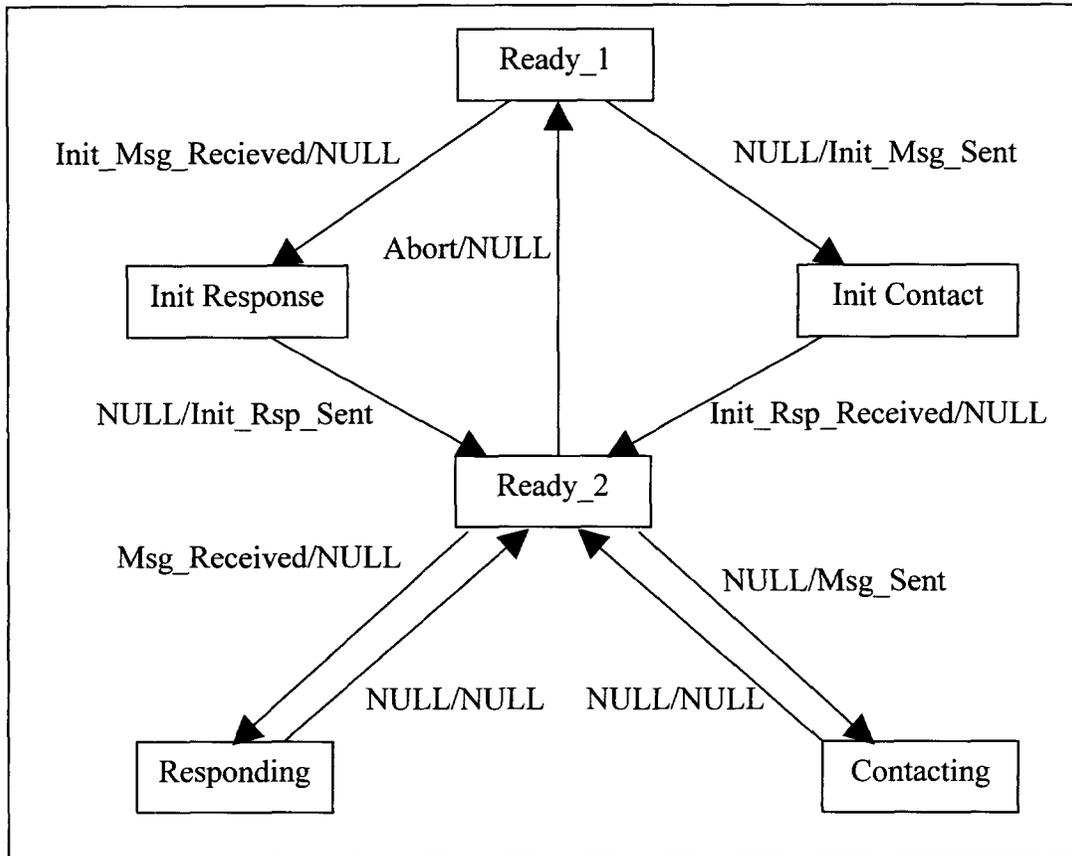
Under the SSL protocol a communicating entity may be either a client or a server in a given communication scenario. The communicating entity that makes first contact automatically becomes a client and the entity that it contacts automatically becomes the server under the SSL protocol. Further, a communicating entity may serve as a client and a server simultaneously if more than one communication scenario is occurring concurrently. Therefore, the first two basic model components that must be combined for model checking is the Client component and the Server component (See Figures 9 and 10 respectively. This combined model component referred to as a “Complete” Communicating Entity in this document. To combine these two model components the union of the two state charts is constructed. (See Figure 11) Multiple instances of this will be model checked together and in conjunction with Client, Server and Simple Communicating Entities and the environment event queue component.



**Figure 9: Client Communication Entity**

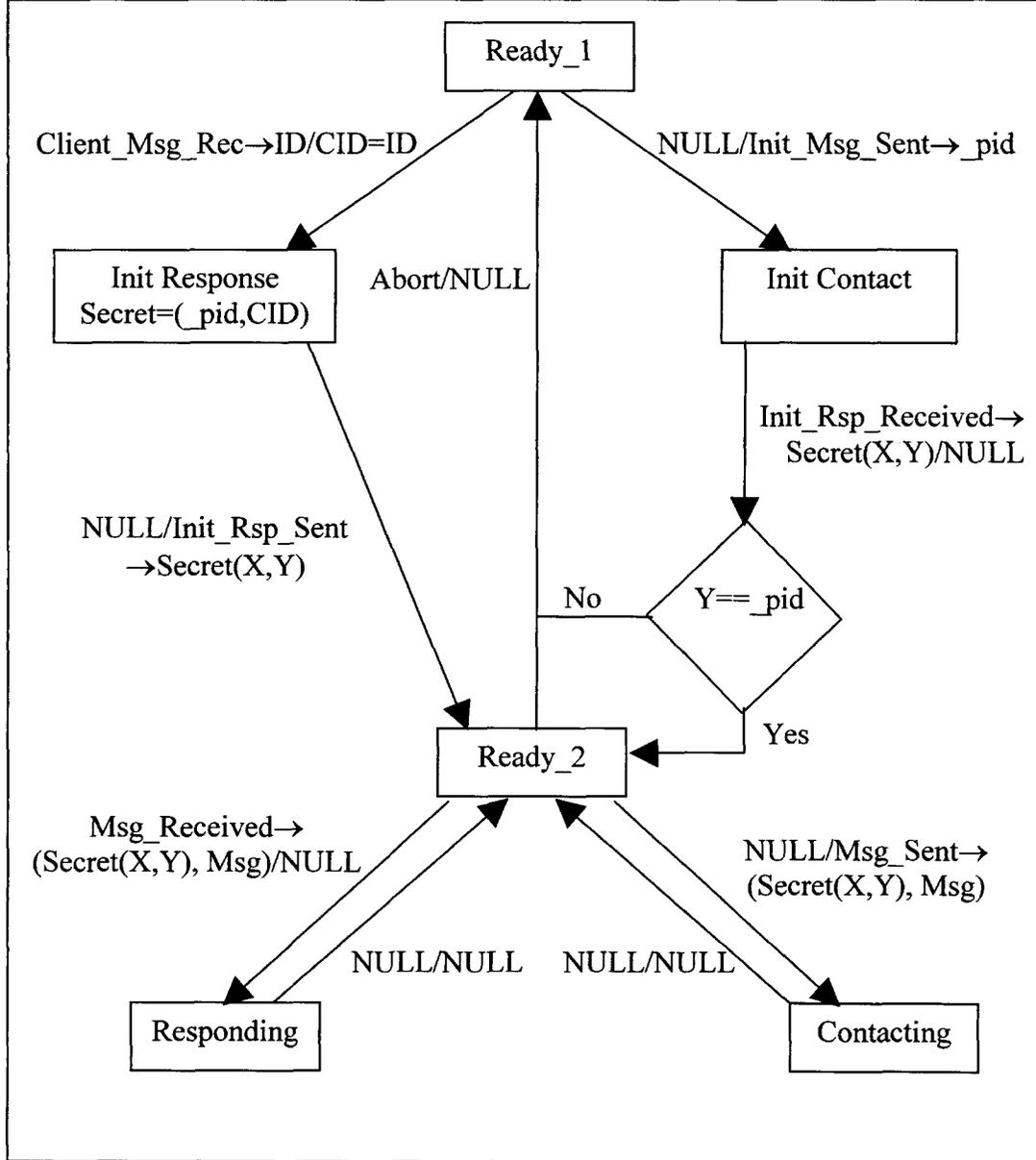


**Figure 10: Server Communication Entity**



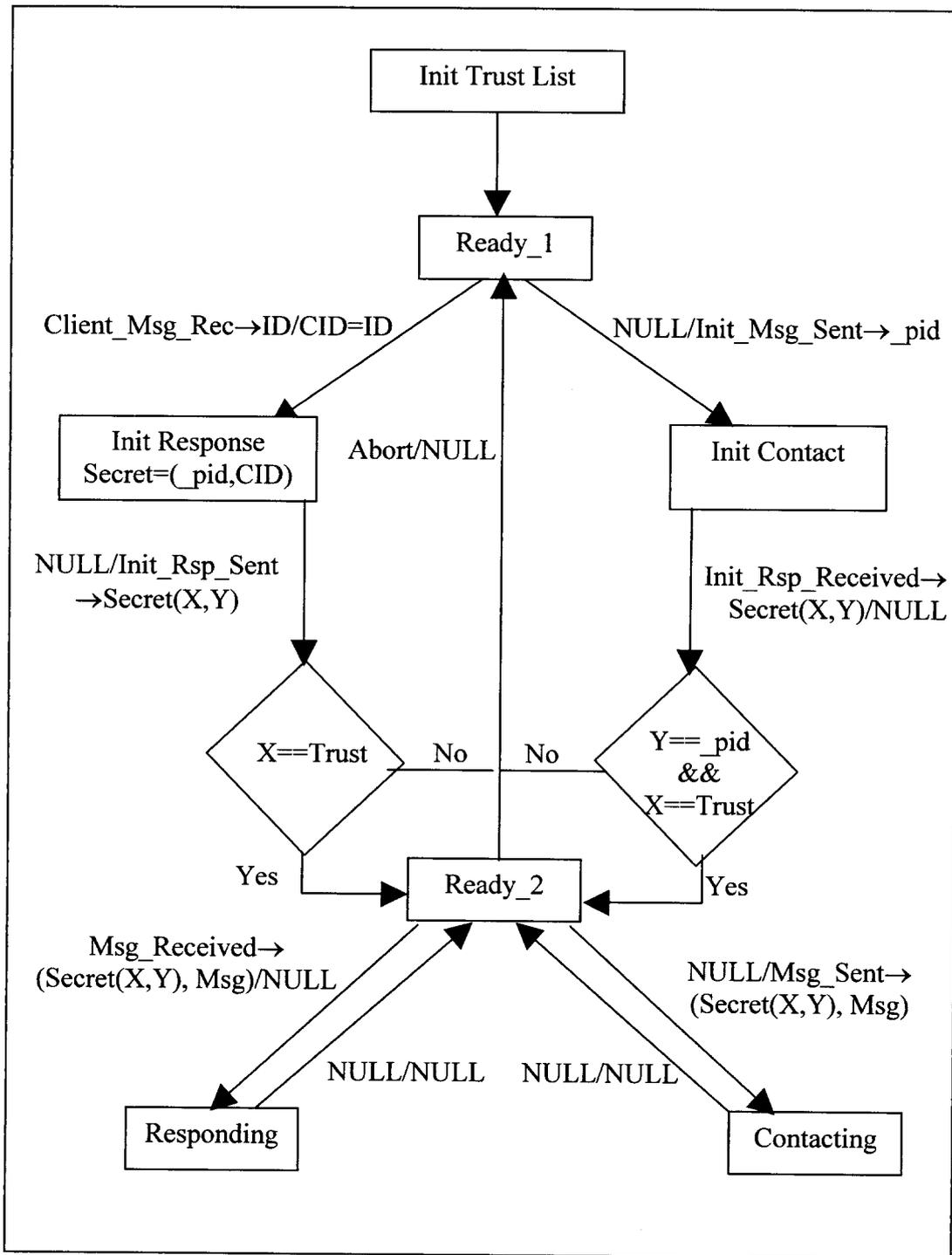
**Figure 11: Complete Communication Entity**

A “Handshake” between two communicating entities is the initialization protocol that they use to set them up for a specific type of ongoing communication. The Complete Communication Entity is combined with the establishment of a secret during the initial handshake between client and server for use in future encryption as per the SSL communication protocol. (See Figure 12) There is no use of certificates to authenticate a communicating entity’s identity at this stage. This entity is the first MCC that substantially contains the characteristics of SSL style communication between two computer systems over a network. This MCC is an abstract representation of public key encryption as used within the SSL communication protocol with timeouts and invalid responses represented by transitions back to the “Ready1” state. In order to make this abstract representation efficient in terms of verifiability the encryption keys can be readily calculated. However, we are assuming that any attack is unable to break encryptions by brute force and can only decrypt encrypted messages if it obtained the required secret. As will be shown in the next section, the attacks are crafted to ignore the ability to predetermine the public key encryption secrets as per the stated assumption. Modeling of full-scale encryption would, by definition of the purpose of encryption, produce a state space that is far too large to search, even by automated means such as the SPIN model checker being used in this verification effort.



**Figure 12: SSL Communication Entity – No ID Check**

The final MCC behavior that does not include an attack is an SSL compliant entity that uses certificates and trust lists to confirm the identity of each SSL communication entity. (See Figure 13) A trust list is simply an internal list of outside communicating entities maintained by a given communicating entity of interest. The list denotes those outside entities with which the entity of interest will allow the exchange of secure information. This is the most secure entity modeled in this research. Thus, any modeled attack that defeats an entity of this type will be cause for investigating a flaw in the SSL protocol.



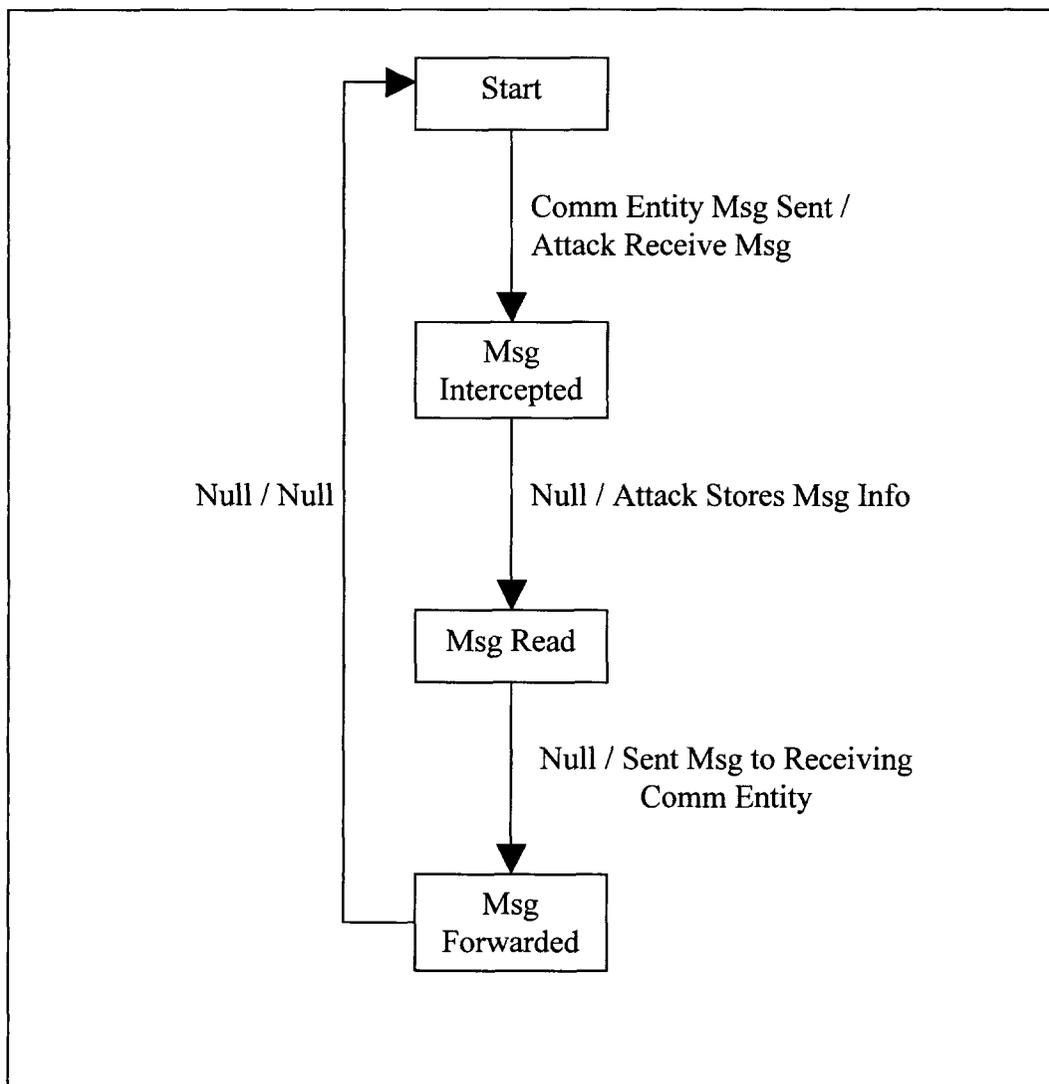
**Figure 13: SSL Communication Entity with Certificates**

The remaining model components provide three attack behaviors:

- Man in the middle attack (See Figure 14)
- Replay attack (See Figure 15)
- Denial of Service (DoS) attack (See figure 16)

Each attack will be combined with the three significant communication MCCs, which are the Complete Communication Entities, the SSL entities without certificates and the SSL entities that use certificates. These form nine MCCs that must be verified with respect to the correctness properties defined for the system (See next section)

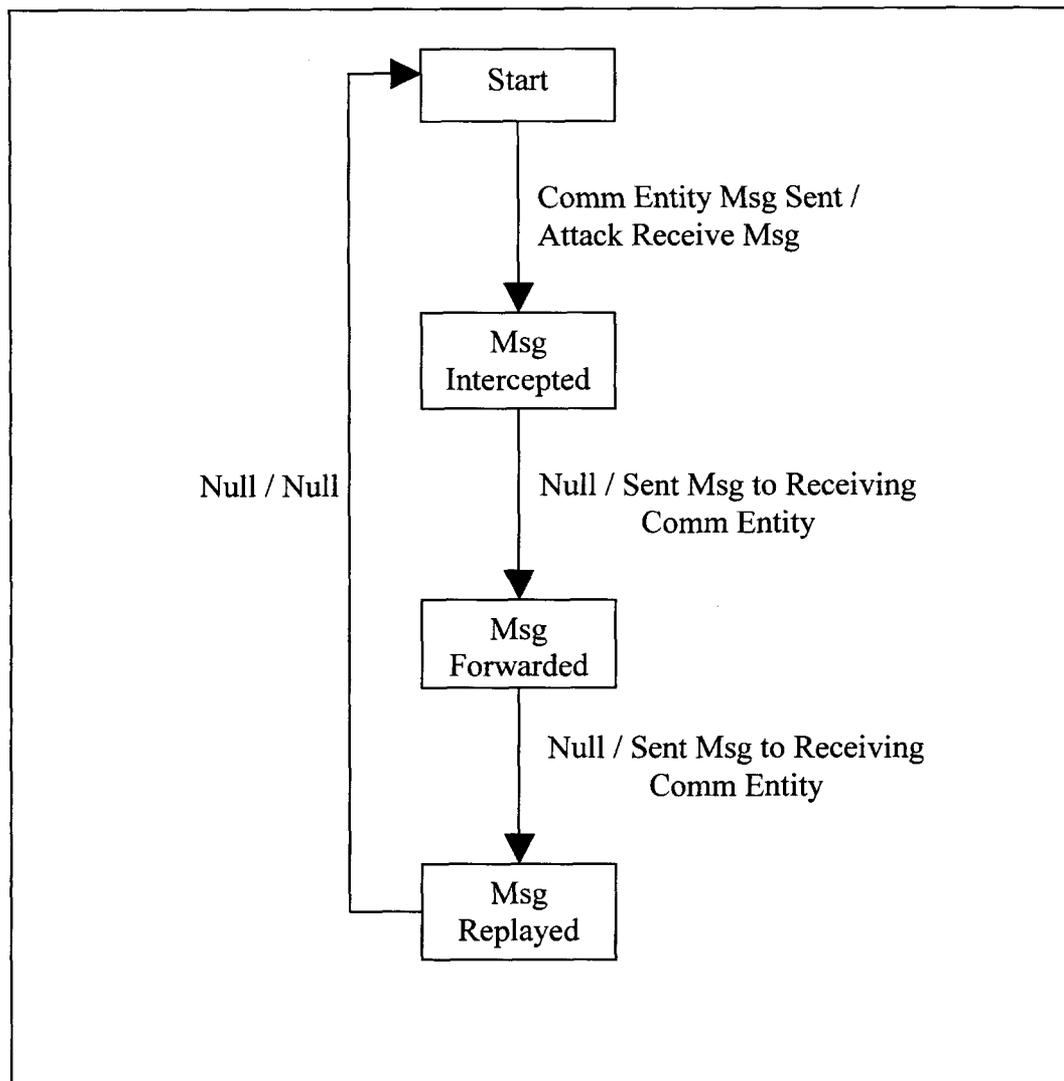
The man in the middle attack intercepts a message coming from one entity and then attempts to pass it on to the intended receiving entity undetected.



**Figure 14: Man-in-the-Middle Attack**

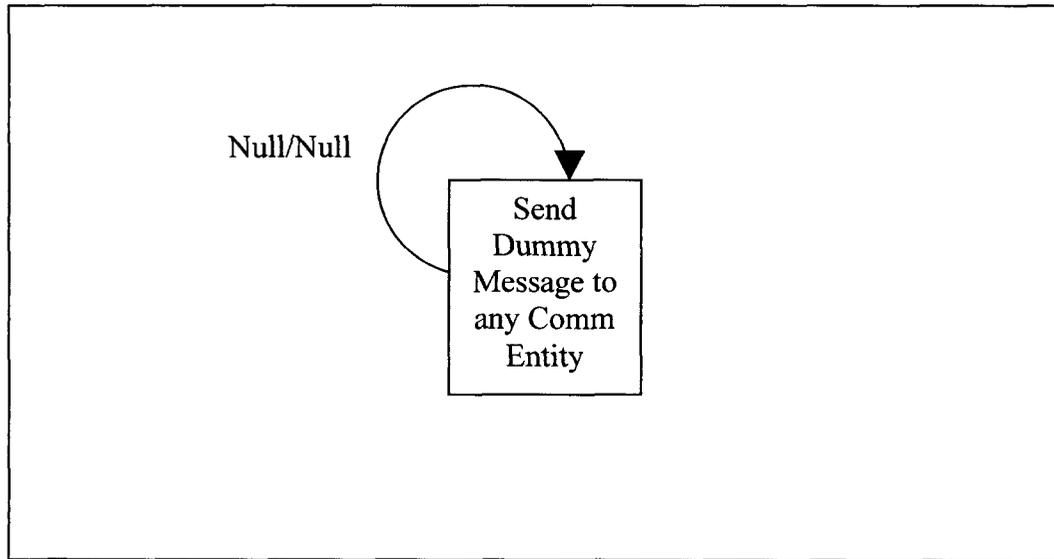
The replay attack (See Figure 15) attempts to intercept a message and forward it on undetected much the same way that the Man in the Middle attack does. In addition, the replay attack stores the message and replays it at a later time to disrupt secure communications. This disruption can take on one of two forms:

- Produce an invalid sequence during the handshake repeatedly to prohibit secure communication from being established.
- Intervene with a message that will confuse secure communications and result in the securely communicating entities revealing their encryption secret(s).



**Figure 15: Replay Attack**

The DoS attack (See figure 16) simply attempts to flood the communication channels with spurious message that will prohibit communication entities from establishing or maintaining secure communication. Thus the authorized entities will denied access to secure communication facilities.



**Figure 16: Denial of Service (DoS) Attack**

## 4. Results of SSL Communication Protocol Verification

The four correctness properties of interest in this verification effort, stated as requirements, are as follows:

1. The SSL secure communication shall initialize eventually unless less an attack has successfully inserted itself in such a manner that the resulting secure communication will be compromised.
2. Once secure communication is established secure contacts and responses will always be reaced.
3. A secure message that has been intercepted shall be detected and not accepted by the SSL communicating recipient of the secure message.
4. Under the rules for attacks an attack may only read messages that are either unsecured or secured messages if the secret has previously been captured. Securely communicating entities shall not reveal their secret even during the handshake initialization

Figure 17 shows a table of verification results over nine variations of the system being model checked. The property states, in terms of the model variables (See excepts in Appendix A) that the state of both of the given entities with eventually reach a state such that they are communicating securely with one another in light of the given attack. The table provides verification for properties 1 though 3.

<b>Each communicating entity will eventually achieve and execute the exchange of secure communication</b>	<b>No Attack</b>	<b>Man in the Middle Attack</b>	<b>Replay Attack</b>	<b>DoS attack</b>
<b>Signed SSL Entities (Certificates)</b>	No Violation	Violation	Violation	Violation
<b>Unsigned SSL Entities (No Certificates)</b>	No Violation	No Violation	Violation	Violation
<b>Non-SSL Client Server Entities</b>	No Violation	No Violation	Violation	Violation

**Figure 17: Verification of Eventually Acheiveing Secure Communication in the Face of Attacks with SSL Entities that use Certificates – Violation means the acctack was recognized.**

The correctness property stated in Figure 17 is expressed as the following LTL property for use by the SPIN model checker.

$$(\langle \langle \text{Entity\_State1} == \text{Contacting} \parallel \text{Entity\_State1} == \text{Responding} \rangle \rangle) \ \&\& \\ (\langle \langle \text{Entity\_State2} == \text{Contacting} \parallel \text{Entity\_State2} == \text{Responding} \rangle \rangle)$$

The “No Attack” column gives a baseline on the behavior of the MCC represented by each row before an attack is added. The fact that there is no violation in any of the attacks verifies that in the absence of the insertion of a successful attack the communicating entities do indeed initialize and communicate. In the remaining three columns a successful attack is inserted into the three different types of communicating entities. A violation confirms that the attack was successful and that the communicating entities detected the attack and stopped before revealing secure communication. Since each column has at least one violation for each attack and the attack is identical across each MCC represented by the rows in the table it can be concluded that the attacks are successful. Both the Replay and DoS attacks and communication is stopped before secure communications are revealed. However, the Man in the Middle attack can go undetected in non-SSL communication and in SSL communications where certificates are not used. It is only when certificates are checked against a trust list for each of the securely communicating entities that the Man in the Middle attack is detected and communication is stopped before revealing the secure communications.

Property 4 was verified by instrumenting the Man in the Middle attack and the environment to make the secret generated by SSL communicating entities, with and without certificates, known to the attack for comparison purposes only. The Man in the Middle attack begins intercepting and examining messages. After the examination the encryption secret in the message is compared to the control secret to determine if the attack has read the secret before encryption takes place. Once the handshake successfully transitions to secure communication the messages are considered encrypted and cannot be read by the attack as per the preface in property requirement 4. This property does not apply to any attack perpetrated on a non-SSL communicating entity since no secret is generated and therefore cannot be intercepted. The DoS and Replay attacks do not attempt to read a message. Therefore neither attack has an opportunity to capture the unencrypted message with the secret even if it is available. With regard to the Man in the Middle attack on SSL entities (with and without certificates), the secret was not successfully captured in either case.

## 5. Conclusion



## References

- [1] D. Gilliam, J. Kelly, M. Bishop, "Reducing Software Security Risk Through an Integrated Approach," Proc. of the Ninth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (June, 2000), Gaithersburg, MD, pp.141-146.
- [2] G. Fink, M. Bishop, "Property Based Testing: A New Approach to Testing for Assurance," ACM SIGSOFT Software Engineering Notes 22(4) (July 1997).
- [3] M. Bishop, "Vulnerabilities Analysis," Proceedings of the Recent Advances in Intrusion Detection (Sep. 1999).
- [4] J. Dodson, "Specification and Classification of Generic Security Flaws for the Tester's Assistant Library," M.S. Thesis, Department of Computer Science, University of California at Davis, Davis CA (June 1996).
- [5] D. Gilliam, J. Kelly, J. Powell, M. Bishop, "Development of a Software Security Assessment Instrument to Reduce Software Security Risk" Proc. of the Tenth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, Boston, MA, pp 144-149.
- [6] D. Gilliam, J. Powell, J. Kelly, M. Bishop, "Reducing Software Security Risk Through an Integrated Approach", IEEE Goddard 26th Annual Software Engineering Workshop
- [7] W. Wen and F Mizoguchi. Model checking Security Protocols: A Case Study Using SPIN, IMC Technical Report, November, 1998.
- [8] J. R. Callahan, S. M. Easterbrook and T. L. Montgomery, "Generating Test Oracles via Model Checking," NASA/WVU Software Research Lab, Fairmont, WV, Technical Report # NASA-IVV-98-015, 1998.
- [9] P. E. Ammann, P. E. Black and W. Majurski. "Using Model Checking to Generate Test Specifications," 2<sup>nd</sup> International Conference on Formal Engineering Methods (1998) pp. 46-54.
- [10] G. Holzmann. Design and Validation of Computer Protocols. Prentice Hall 1990; ISBN: 0135399254 .

## Appendix A: Promela Code Excerpts

```
mtype {Ready, Responding, Contacting};

mtype Entity_State1 = Ready;
mtype Entity_State2 = Ready;
bit Msg_Received = 0;      /* 0 = message not received -- 1= message received
*/
bit Msg_Sent = 0; /* 0 = message not sent -- 1 = message sent */
int Msg_Id = -1; /* -1 = no message Id */
chan Network1 = [1] of {bit};
chan Network2 = [1] of {bit}

active proctype CommEntity1()
{
    do
        ::Entity_State1 == Ready -> /* Prerequisite State */
            if
                :: Network1?Msg_Received -> /* Prerequisite Event */
                    Entity_State1 = Responding; /* New State Entered */

                ::empty(Network2) ->
                    Network2!1; /* Transition Action */
                    Msg_Sent=1; /* Transition Side Effect */
                    Entity_State1 = Contacting; /* New State Entered */
                    Msg_Sent = 0 /* Side effect does not persist within a state
*/
            fi

        ::Entity_State1 == Contacting -> /* Prerequisite State */
            Entity_State1 = Ready; /* New State Entered */

        ::Entity_State1 == Responding -> /* Prerequisite State */
            Entity_State1 = Ready; /* New State Entered */
    od
}
```

**Figure 8.1: Promela Listing of Simple Communicating Entities**

```

mtype {Ready, Responding, Contacting};

mtype Entity_State1 = Ready;
mtype Entity_State2 = Ready;
bit Msg_Received = 0;      /* 0 = message not received -- 1 = message received
*/
bit Msg_Sent = 0; /* 0 = message not sent -- 1 = message sent */
int Msg_Id = -1; /* -1 = no message Id */
chan Network1 = [1] of {bit};
chan Network2 = [1] of {bit}

... CommEntity1 Model Specification (See Figure 1.1) would be Here ...

active proctype CommEntity2()
{
    do
        ::Entity_State2 == Ready -> /* Prerequisite State */
            if
                :: Network2?Msg_Received -> /* Prerequisite Event */
                    Entity_State2 = Responding; /* New State Entered
*/

                ::empty(Network1) ->
                    Network1!1; /* Transition Action */
                    Msg_Sent=1; /* Transition Side Effect */
                    Entity_State2 = Contacting; /* New State Entered */
                    Msg_Sent = 0 /* Side effect does not persist within a state
*/

            fi

        ::Entity_State2 == Contacting -> /* Prerequisite State */
            Entity_State2 = Ready; /* New State Entered */

        ::Entity_State2 == Responding -> /* Prerequisite State */
            Entity_State2 = Ready; /* New State Entered */
    od
}

```

**Figure 8.2: Promela Listing of Simple Communicating Entities**

```

mtype {Ready1, Ready2, Responding, Contacting};
mtype Entity_State1 = Ready1;
mtype Entity_State2 = Ready1;
bit Msg_Received = 0;      /* 0 = message not received -- 1 = message received
*/
bit Msg_Sent = 0; /* 0 = message not sent -- 1 = message sent */
int Msg_Id = -1; /* -1 = no message Id */
chan Network1 = [1] of {bit};
chan Network2 = [1] of {bit}

active proctype CommEntity1()
{
    do
        ::Entity_State1 == Ready1 ->
            Network2!1; /* Transition Action Init Msg */
            Entity_State1 = Ready2;

        ::Entity_State1 == Ready2 -> /* Prerequisite State */
            if
                :: Network1?Msg_Received -> /* Prerequisite Event */
                    Entity_State1 = Responding; /* New State Entered */
                ::empty(Network2) ->
                    Network2!1; /* Transition Action */
                    Msg_Sent=1; /* Transition Side Effect */
                    Entity_State1 = Contacting; /* New State Entered */
                    Msg_Sent = 0 /* Side effect does not persist w/in a state */
                ::skip -> /* Placeholder for Abort Event */
                    Entity_State1 = Ready1;
            fi

        ::Entity_State1 == Contacting -> /* Prerequisite State */
            Entity_State1 = Ready2; /* New State Entered */

        ::Entity_State1 == Responding -> /* Prerequisite State */
            Entity_State1 = Ready2; /* New State Entered */
    od
}

```

**Figure 9.1: Promela Listing of Client Communicating Entity**

```

mtype {Ready1, Ready2, Responding, Contacting};
mtype Entity_State1 = Ready1;
mtype Entity_State2 = Ready1;
bit Msg_Received = 0; /* 0 = message not received -- 1 = message received */
bit Msg_Sent = 0; /* 0 = message not sent -- 1 = message sent */
int Msg_Id = -1; /* -1 = no message Id */
chan Network1 = [1] of {bit};
chan Network2 = [1] of {bit}

... Comm Entity 1 Model Specification would go Here ...

active proctype CommEntity2()
{
    do
        ::Entity_State2 == Ready1 ->
            Network2?Msg_Received -> /* Prerequisite Init Event */
            Entity_State2 = Ready2
        ::Entity_State2 == Ready2 -> /* Prerequisite State */
            if
                :: Network2?Msg_Received -> /* Prerequisite Event */
                    Entity_State2 = Responding; /* New State Entered */
                    ::empty(Network1) ->
                        Network1!!1; /* Transition Action */
                        Msg_Sent=1; /* Transition Side Effect */
                        Entity_State2 = Contacting; /* New State Entered */
                        Msg_Sent = 0 /* Side effect does not persist w/in a state */
                    ::skip -> /* Placeholder for Abort Event */
                        Entity_State2 = Ready1;
            fi
        ::Entity_State2 == Contacting -> /* Prerequisite State */
            Entity_State2 = Ready2; /* New State Entered */
        ::Entity_State2 == Responding -> /* Prerequisite State */
            Entity_State2 = Ready2; /* New State Entered */
    od
}

```

**Figure 10.1: Promela Listing of Server Communicating Entity**

```

mtype {Ready1, Ready2, InitContact, InitResponse, Responding, Contacting};
mtype Entity_State1 = Ready1;
mtype Entity_State2 = Ready1;
int Msg_Received1=0; /* 0 = msg not rec. -- 1= msg rec. -- 2=init msg rec. */
int Msg_Received2=0; /* 0 = msg not rec. -- 1= msg rec. -- 2=init msg rec. */
bit Msg_Sent = 0; /* 0 = message not sent -- 1 = message sent */
int Msg_Id = -1; /* -1 = no message Id */
chan Network1 = [1] of {int};
chan Network2 = [1] of {int}
active proctype CommEntity1()
{
    do
        ::Entity_State1 == Ready1 ->
            if
                ::Network1?Msg_Received1 ->
                    if
                        ::Msg_Received1 == 2-> /* Prerequisite Event */
                            Entity_State1 = InitResponse;
                        ::Msg_Received1 != 2 ->
                            Entity_State1 = Ready1; /* Response Aborted */
                    fi
                ::empty(Network2) ->
                    Network2!2; /* Transition Action Init Msg */
                    Entity_State1 = InitContact;
            fi
        ::Entity_State1 == InitContact ->
            Network1?Msg_Received1;
            if
                ::Msg_Received1 == 2 ->
                    Entity_State1 = Ready2; /* Comm Initiated */
                ::Msg_Received1 != 2 ->
                    Entity_State1 = Ready1; /* Comm Aborted */
            fi
        ::Entity_State1 == InitResponse ->
            Network2!2; /* Transition Action Init Msg */
            Entity_State1 = Ready2; /* Comm Initiated */
        ::Entity_State1 == Ready2 -> /* Prerequisite State */
            if
                :: Network1?Msg_Received1 -> /* Prerequisite Event */
                    Entity_State1 = Responding; /* New State Entered */
                ::empty(Network2) ->
                    Network2!1; /* Transition Action */
                    Msg_Sent=1; /* Transition Side Effect */
                    Entity_State1 = Contacting; /* New State Entered */
                    Msg_Sent = 0 /* Side effect does not persist within a state */
                ::skip -> /* End Comm Session by Aborting */
                    Entity_State1 = Ready1;
            fi
        ::Entity_State1 == Contacting -> /* Prerequisite State */
            Entity_State1 = Ready2; /* New State Entered */
        ::Entity_State1 == Responding -> /* Prerequisite State */
            Entity_State1 = Ready2; /* New State Entered */
    od
}

```

**Figure 11.1: Promela Listing of Client/Server Communicating Entity**

```

mtype {Ready1, Ready2, InitContact, InitResponse, Responding, Contacting};
mtype Entity_State1 = Ready1;
mtype Entity_State2 = Ready1;
int Msg_Received1=0; /* 0 = msg not rec. -- 1= msg rec. -- 2=init msg rec. */
int Msg_Received2=0; /* 0 = msg not rec. -- 1= msg rec. -- 2=init msg rec. */
bit Msg_Sent = 0; /* 0 = message not sent -- 1 = message sent */
int Msg_Id = -1; /* -1 = no message Id */
chan Network1 = [1] of {int};
chan Network2 = [1] of {int}

... Communication Entity 1 would go Here ...

active proctype CommEntity2()
{
    do
        ::Entity_State2 == Ready1 ->
            if
                ::Network2?Msg_Received2 ->
                    if
                        ::Msg_Received2 == 2-> /* Prerequisite Event */
                            Entity_State2 = InitResponse;
                        ::Msg_Received2!= 2 ->
                            Entity_State2 = Ready1; /* Response Aborted */
                    fi
                ::empty(Network1) ->
                    Network1!2; /* Transition Action Init Msg */
                    Entity_State2 = InitContact;
            fi
        ::Entity_State2 == InitContact ->
            Network2?Msg_Received2;
            if
                ::Msg_Received2 == 2 ->
                    Entity_State2 = Ready2; /* Comm Initiated */
                ::Msg_Received2!= 2 ->
                    Entity_State2 = Ready1; /* Comm Aborted */
            fi
        ::Entity_State2 == InitResponse ->
            Network1!2; /* Transition Action Init Msg */
            Entity_State2 = Ready2; /* Comm Initiated */
        ::Entity_State2 == Ready2 -> /* Prerequisite State */
            if
                :: Network2?Msg_Received2 -> /* Prerequisite Event */
                    Entity_State2 = Responding; /* New State Entered */
                ::empty(Network1) ->
                    Network1!1; /* Transition Action */
                    Msg_Sent=1; /* Transition Side Effect */
                    Entity_State2 = Contacting; /* New State Entered */
                    Msg_Sent = 0 /* Side effect does not persist within a state */
                ::skip -> /* End Comm Session by Aborting */
                    Entity_State2 = Ready1;
            fi
        ::Entity_State2 == Contacting -> /* Prerequisite State */
            Entity_State2 = Ready2; /* New State Entered */
        ::Entity_State2 == Responding -> /* Prerequisite State */
            Entity_State2 = Ready2; /* New State Entered */
    od
}

```

**Figure 11.2: Promela Listing of Client/Server Communicating Entity**

```

mtype {Ready1, Ready2, InitContact, InitResponse, Responding, Contacting};
mtype Entity_State1 = Ready1;
mtype Entity_State2 = Ready1;
int Msg_Received1=0; /* 0 = message not received -- 1= message received -- 2=initi message received */
int Msg_Received2=0; /* 0 = message not received -- 1= message received -- 2=initi message received */
bit Msg_Sent = 0; /* 0 = message not sent -- 1 = message sent */
int Msg_Id = -1; /* -1 = no message Id */
chan Network1 = [1] of {int,int,int};
chan Network2 = [1] of {int,int,int};
active proctype CommEntity1()
{
    int SecretX=-1;
    int SecretY=-1;
    int CID = -1; /* Client ID */
    do
        ::Entity_State1 == Ready1 ->
            if
                ::Network1?SecretX,SecretY,Msg_Received1 ->
                    if
                        ::SecretY== -1 -> /* Prerequisite Event */
                            CID=Msg_Received1;
                            Entity_State1 = InitResponse;
                        ::SecretY != -1 ->
                            Entity_State1 = Ready1; /* Response Aborted */
                    fi
                ::skip ->
                    Network2!-1,-1,_pid; /* Transition Action Init Msg */
                    Entity_State1 = InitContact;
            fi
        ::Entity_State1 == InitContact ->
            Network1?SecretX,SecretY,Msg_Received1;
            if
                ::SecretY == _pid ->
                    Entity_State1 = Ready2; /* Comm Initiated */
                ::SecretY != _pid ->
                    Entity_State1 = Ready1; /* Comm Aborted */
            fi
        ::Entity_State1 == InitResponse ->
            Network2!_pid,CID,2; /* Transition Action Init Msg */
            Entity_State1 = Ready2; /* Comm Initiated */
        ::Entity_State1 == Ready2 -> /* Prerequisite State */
            if
                ::empty(Network1) ->
                    Network1?SecretX,SecretY,Msg_Received1 -> /* Prerequisite Event */
                    Entity_State1 = Responding; /* New State Entered */
                ::skip ->
                    Network2!_pid,CID,1; /* Transition Action */
                    Msg_Sent=1; /* Transition Side Effect */
                    Entity_State1 = Contacting; /* New State Entered */
                    Msg_Sent = 0 /* Side effect does not persist within a state */
                ::empty(Network1) -> /* End Comm Session by Aborting */
                    Entity_State1 = Ready1;
            fi
        ::Entity_State1 == Contacting -> /* Prerequisite State */
            Entity_State1 = Ready2; /* New State Entered */
        ::Entity_State1 == Responding -> /* Prerequisite State */
            Entity_State1 = Ready2; /* New State Entered */
    od
}

```

**Figure 12.1: SSL (No ID Check) Communicating Entity 1**

```

active proctype CommEntity2()
{
    int SecretX=-1;
    int SecretY=-1;
    int CID = -1; /* Client ID */
    do
        ::Entity_State1 == Ready1 ->
            if
                ::Network2?SecretX,SecretY,Msg_Received2 ->
                    if
                        ::SecretY== -1 -> /* Prerequisite Event */
                            CID=Msg_Received2;
                            Entity_State1 = InitResponse;
                        ::SecretY != -1 ->
                            Entity_State1 = Ready1; /* Response Aborted */
                    fi
                ::skip ->
                    Network1!-1,-1,_pid; /* Transition Action Init Msg */
                    Entity_State1 = InitContact;
            fi

        ::Entity_State1 == InitContact ->
            Network2?SecretX,SecretY,Msg_Received2;
            if
                ::SecretY == _pid ->
                    Entity_State1 = Ready2; /* Comm Initiated */
                ::SecretY != _pid ->
                    Entity_State1 = Ready1; /* Comm Aborted */
            fi

        ::Entity_State1 == InitResponse ->
            Network1!_pid,CID,2; /* Transition Action Init Msg */
            Entity_State1 = Ready2; /* Comm Initiated */

        ::Entity_State1 == Ready2 -> /* Prerequisite State */
            if
                ::nempty(Network1) ->
                    Network2?SecretX,SecretY,Msg_Received2 -> /* Prerequisite Event */
                    Entity_State1 = Responding; /* New State Entered */
                ::skip ->
                    Network1!_pid,CID,1; /* Transition Action */
                    Msg_Sent=1; /* Transition Side Effect */
                    Entity_State1 = Contacting; /* New State Entered */
                    Msg_Sent = 0 /* Side effect does not persist within a state */
                ::empty(Network1) -> /* End Comm Session by Aborting */
                    Entity_State1 = Ready1;
            fi

        ::Entity_State1 == Contacting -> /* Prerequisite State */
            Entity_State1 = Ready2; /* New State Entered */

        ::Entity_State1 == Responding -> /* Prerequisite State */
            Entity_State1 = Ready2; /* New State Entered */
    od
}

```

**Figure 12.2: SSL (No ID Check) Communicating Entity 2**

```

...
active proctype CommEntity1()
{
    int SecretX=-1;
    int SecretY=-1;
    int CID = -1; /* Client ID */
    int Trust = -1; /* Single certificate Trust list */
    atomic{
        Network2!-1,-1,_pid;
        Network1?SecretX,SecretY,Trust;}
    do
        ::Entity_State1 == Ready1 ->
            if
                ::Network1?SecretX,SecretY,Msg_Received1 ->
                    if
                        ::SecretY== -1 -> /* Prerequisite Event */
                            CID=Msg_Received1;
                            Entity_State1 = InitResponse;
                        :: SecretY != -1 ->
                            Entity_State1 = Ready1; /* Response Aborted */
                    fi
                ::skip ->
                    Network2!-1,-1,_pid; /* Transition Action Init Msg */
                    Entity_State1 = InitContact;
            fi
        ::Entity_State1 == InitContact ->
            Network1?SecretX,SecretY,Msg_Received1;
            if
                ::SecretX == Trust && SecretY == _pid ->
                    Entity_State1 = Ready2; /* Comm Initiated */
                ::SecretX != Trust || SecretY != _pid ->
                    Entity_State1 = Ready1; /* Comm Aborted */
            fi
        ::Entity_State1 == InitResponse ->
            if
                ::Msg_Received1 == Trust ->
                    Network2!_pid,CID,2; /* Transition Action Init Msg */
                    Entity_State1 = Ready2; /* Comm Initiated */
                ::Msg_Received1 != Trust ->
                    Entity_State1 = Ready1;
            fi
        ::Entity_State1 == Ready2 -> /* Prerequisite State */
            if
                :: nempty(Network1) ->
                    Network1?SecretX,SecretY,Msg_Received1 -> /* Prerequisite Event */
                    Entity_State1 = Responding; /* New State Entered */
                ::empty(Network1) ->
                    Network2!_pid,CID,1; /* Transition Action */
                    Msg_Sent=1; /* Transition Side Effect */
                    Entity_State1 = Contacting; /* New State Entered */
                    Msg_Sent = 0 /* Side effect does not persist within a state */
            fi
        ::Entity_State1 == Contacting -> /* Prerequisite State */
            Entity_State1 = Ready2; /* New State Entered */
            Network1?SecretX,SecretY,Msg_Received1;
        ::Entity_State1 == Responding -> /* Prerequisite State */
            Entity_State1 = Ready2; /* New State Entered */
            Network2!_pid,CID,1;
    od
}

```

**Figure 13.1 SSL Communication Entity 1**

```

active proctype CommEntity2()
{
    int SecretX=-1;
    int SecretY=-1;
    int CID = -1; /* Client ID */
    int Trust = -1; /* Single Certificate Trust List */
    atomic{
        Network1!-1,-1,_pid;
        Network2?SecretX,SecretY,Trust;}
    do
        ::Entity_State2 == Ready1 ->
            if
                ::Network2?SecretX,SecretY,Msg_Received2 ->
                    if
                        ::SecretY==-1 -> /* Prerequisite Event */
                            CID=Msg_Received2;
                            Entity_State2 = InitResponse;
                        ::SecretY != -1 ->
                            Entity_State2 = Ready1; /* Response Aborted */
                    fi
                ::skip ->
                    Network1!-1,-1,_pid; /* Transition Action Init Msg */
                    Entity_State2 = InitContact;
            fi
        ::Entity_State2 == InitContact ->
            Network2?SecretX,SecretY,Msg_Received2;
            if
                ::SecretX == Trust && SecretY == _pid ->
                    Entity_State2 = Ready2; /* Comm Initiated */
                ::SecretX != Trust || SecretY != _pid ->
                    Entity_State2 = Ready1; /* Comm Aborted */
            fi
        ::Entity_State2 == InitResponse ->
            if
                ::Msg_Received2 == Trust ->
                    Network1!_pid,CID,2; /* Transition Action Init Msg */
                    Entity_State2 = Ready2; /* Comm Initiated */
                ::Msg_Received2 != Trust ->
                    Entity_State2 = Ready1;
            fi
        ::Entity_State2 == Ready2 -> /* Prerequisite State */
            if
                ::nempty(Network1) ->
                    Network2?SecretX,SecretY,Msg_Received2 -> /* Prerequisite Event */
                    Entity_State2 = Responding; /* New State Entered */
                ::empty(Network1) ->
                    Network1!_pid,CID,2; /* Transition Action */
                    Msg_Sent=1; /* Transition Side Effect */
                    Entity_State2 = Contacting; /* New State Entered */
                    Msg_Sent = 0 /* Side effect does not persist within a state */
            fi
        ::Entity_State2 == Contacting -> /* Prerequisite State */
            Entity_State2 = Ready2; /* New State Entered */
            Network2?SecretX,SecretY,Msg_Received2;
        ::Entity_State2 == Responding -> /* Prerequisite State */
            Entity_State2 = Ready2; /* New State Entered */
            Network1!_pid,CID,2;
    od
}

```

**Figure 13.2: SSL Communication Entity 2**

```

active proctype ManInMiddleAttack()
{
    int Sender = -1;
    int Receiver = -1;
    int Message = -1;

    progress: do
        ::nempty(Network1) && (Sender == -1 || Receiver == -1)->
            atomic
            {
                Network1?Sender,Receiver,Message;
                Network1!_pid,Receiver,Message;
                empty(Network1);
            }
        ::nempty(Network2) && (Sender == -1 || Receiver == -1)->
            atomic
            {
                Network2?Sender,Receiver,Message;
                Network2!_pid,Receiver,Message;
                empty(Network2);
            }
    od
}

```

**Figure 14.1: Man-in-the-Middle Attack**

```

active proctype ReplayAttack()
{
    int Sender = -1;
    int Receiver = -1;
    int Message = -1;

    do
        ::nempty(Network1) ->
            Network1?Sender,Receiver,Message;
            Network1!_pid,Receiver,Message;
            Network1!_pid,Receiver,Message;

        ::nempty(Network2) ->
            Network2?Sender,Receiver,Message;
            Network2!_pid,Receiver,Message;
            Network2!_pid,Receiver,Message;
    od
}

```

**Figure 15.1: Replay Attack**

```
active proctype DoSAttack()
{
    int Sender = -1;
    int Receiver = -1;
    int Message = -1;

    do
        ::Network1!_pid,-1,-1;
        ::Network2!_pid,-1,-1;
    od
}
```

**Figure 16.1: Denial of Service (DoS) Attack**

## Appendix B: SPIN Output for Verification Results

```
#define p Entity_State1==Contacting
#define q Entity_State1==Responding
#define r Entity_State2==Contacting
#define s Entity_State2==Responding

(<> (p || q)) && (<> (r || s))

/*
 * Formula As Typed: (<> (p || q)) && (<> (r || s))
 * The Never Claim Below Corresponds
 * To The Negated Formula !((<> (p || q)) && (<> (r || s)))
 * (formalizing violations of the original)
 */

never { /* !((<> (p || q)) && (<> (r || s))) */
accept_init:
T0_init:
    if
    :: (! ((r)) && ! ((s))) -> goto accept_S2
    :: (! ((p)) && ! ((q))) -> goto accept_S5
    fi;
accept_S2:
T0_S2:
    if
    :: (! ((r)) && ! ((s))) -> goto accept_S2
    fi;
accept_S5:
T0_S5:
    if
    :: (! ((p)) && ! ((q))) -> goto accept_S5
    fi;
}
```

**Figure 17.1: Property Expressing “Eventually Secure Communication is Achieved”**

warning: -i or -I work for safety properties only  
warning: for p.o. reduction to be valid the never claim must be stutter-invariant  
(never claims generated from LTL formulae are stutter-invariant)  
(Spin Version 4.0.4 -- 12 April 2003)  
+ Partial Order Reduction

Full statespace search for:

never-claim +  
assertion violations + (if within scope of claim)  
acceptance cycles + (fairness disabled)  
invalid endstates - (disabled by never-claim)

State-vector 96 byte, depth reached 65, **errors: 0**

688 states, stored (1376 visited)

1402 states, matched

2778 transitions (= visited+matched)

96 atomic steps

hash conflicts: 4 (resolved)

(max size  $2^{19}$  states)

2.622 memory usage (Mbyte)

unreached in proctype CommEntity1

line 64, state 32, "Entity\_State1 = Ready1"

line 86, state 55, "-end-"

(2 of 55 states)

unreached in proctype CommEntity2

line 138, state 32, "Entity\_State2 = Ready1"

line 161, state 55, "-end-"

(2 of 55 states)

**Figure 17.2: Property verifies for SSL Signed Entities when no attack is present  
– No Attack was recognized**

```

preparing trail, please wait...done
spin: warning, "pan_in", global, 'bit Msg_Sent' variable is never used
spin: warning, "pan_in", global, 'int Msg_Id' variable is never used
spin: couldn't find claim (ignored)
2:      proc 1 (CommEntity2) line 97 "pan_in" (state -)      [values: 1!-1,-1,1]
2:      proc 1 (CommEntity2) line 97 "pan_in" (state 1)     [Network1!-(1),-(1),_pid]
4:      proc 2 (ManInMiddleAttack) line 170 "pan_in" (state 1)  [!(empty(Network1))&&((Sender==
(1))|(Receiver==(1))))]
6:      proc 2 (ManInMiddleAttack) line 173 "pan_in" (state -)      [values: 1?-1,-1,1]
6:      proc 2 (ManInMiddleAttack) line 173 "pan_in" (state 2)     [Network1?Sender,Receiver,Message]
7:      proc 2 (ManInMiddleAttack) line 174 "pan_in" (state -)     [values: 1!2,-1,1]
7:      proc 2 (ManInMiddleAttack) line 174 "pan_in" (state 3)     [Network1!_pid,Receiver,Message]
9:      proc 0 (CommEntity1) line 24 "pan_in" (state -)      [values: 2!-1,-1,0]
9:      proc 0 (CommEntity1) line 24 "pan_in" (state 1)     [Network2!-(1),-(1),_pid]
10:     proc 0 (CommEntity1) line 25 "pan_in" (state -)     [values: 1?2,-1,1]
10:     proc 0 (CommEntity1) line 25 "pan_in" (state 2)     [Network1?SecretX,SecretY,Trust]
12:     proc 2 (ManInMiddleAttack) line 175 "pan_in" (state 4)  [!(empty(Network1))]
14:     proc 2 (ManInMiddleAttack) line 178 "pan_in" (state 6)  [!(empty(Network2))&&((Sender==
(1))|(Receiver==(1))))]
16:     proc 2 (ManInMiddleAttack) line 181 "pan_in" (state -)  [values: 2?-1,-1,0]
16:     proc 2 (ManInMiddleAttack) line 181 "pan_in" (state 7)  [Network2?Sender,Receiver,Message]
17:     proc 2 (ManInMiddleAttack) line 182 "pan_in" (state -)  [values: 2!2,-1,0]
17:     proc 2 (ManInMiddleAttack) line 182 "pan_in" (state 8)  [Network2!_pid,Receiver,Message]
19:     proc 1 (CommEntity2) line 98 "pan_in" (state -)      [values: 2?2,-1,0]
19:     proc 1 (CommEntity2) line 98 "pan_in" (state 2)     [Network2?SecretX,SecretY,Trust]
21:     proc 2 (ManInMiddleAttack) line 183 "pan_in" (state 9)  [!(empty(Network2))]
23:     proc 1 (CommEntity2) line 102 "pan_in" (state 4)     [!(Entity_State2==Ready1)]
25:     proc 0 (CommEntity1) line 29 "pan_in" (state 4)     [!(Entity_State1==Ready1)]
27:     proc 0 (CommEntity1) line 42 "pan_in" (state 13)     [!(Entity_State2!=InitContact)]
28:     proc 0 (CommEntity1) line 43 "pan_in" (state -)      [values: 2!-1,-1,0]
28:     proc 0 (CommEntity1) line 43 "pan_in" (state 14)     [Network2!-(1),-(1),_pid] <merge 0 now @15>
28:     proc 0 (CommEntity1) line 44 "pan_in" (state 15)     [Entity_State1 = InitContact] <merge 52 now @52>
30:     proc 2 (ManInMiddleAttack) line 178 "pan_in" (state 6)  [!(empty(Network2))&&((Sender==
(1))|(Receiver==(1))))]
32:     proc 2 (ManInMiddleAttack) line 181 "pan_in" (state -)  [values: 2?-1,-1,0]
32:     proc 2 (ManInMiddleAttack) line 181 "pan_in" (state 7)  [Network2?Sender,Receiver,Message]
33:     proc 2 (ManInMiddleAttack) line 182 "pan_in" (state -)  [values: 2!2,-1,0]
33:     proc 2 (ManInMiddleAttack) line 182 "pan_in" (state 8)  [Network2!_pid,Receiver,Message]
35:     proc 1 (CommEntity2) line 107 "pan_in" (state -)     [values: 2?2,-1,0]
35:     proc 1 (CommEntity2) line 107 "pan_in" (state 5)     [Network2?SecretX,SecretY,Msg_Received2]
36:     proc 1 (CommEntity2) line 112 "pan_in" (state 9)     [!(SecretX!=-(1))|(SecretY!=-(1))] <merge 0 now
@10>
36:     proc 1 (CommEntity2) line 113 "pan_in" (state 10)     [Entity_State2 = Ready1] <merge 52 now @52>
38:     proc 2 (ManInMiddleAttack) line 183 "pan_in" (state 9)  [!(empty(Network2))]
40:     proc 1 (CommEntity2) line 102 "pan_in" (state 4)     [!(Entity_State2==Ready1)]
42:     proc 0 (CommEntity1) line 49 "pan_in" (state 19)     [!(Entity_State1==InitContact)]
<<<<<START OF CYCLE>>>>>
45:     proc 0 (CommEntity1) line 73 "pan_in" (state -)      [values: 2!0,-1,1]
45:     proc 0 (CommEntity1) line 73 "pan_in" (state 40)     [Network2!_pid,CID,1]
spin: trail ends after 45 steps
#processes: 3
45:     proc 2 (ManInMiddleAttack) line 169 "pan_in" (state 11)
45:     proc 1 (CommEntity2) line 103 "pan_in" (state 18)
45:     proc 0 (CommEntity1) line 74 "pan_in" (state 41)
3 processes created

```

**Figure 17.3: Property Counterexample for SSL Signed Entities - Man in the Middle Attack is recognized and secure communication is stopped**

```

preparing trail, please wait...done
spin: warning, "pan_in", global, 'bit Msg_Sent' variable is never used
spin: warning, "pan_in", global, 'int Msg_Id' variable is never used
spin: couldn't find claim (ignored)
2:   proc 1 (CommEntity2) line 97 "pan_in" (state -) [values: 1!-1,-1,1]
2:   proc 1 (CommEntity2) line 97 "pan_in" (state 1) [Network1!-(1),-(1),_pid]
4:   proc 2 (ReplayAttack) line 170 "pan_in" (state 1) [(nempty(Network1))]
6:   proc 2 (ReplayAttack) line 171 "pan_in" (state -) [values: 1?-1,-1,1]
6:   proc 2 (ReplayAttack) line 171 "pan_in" (state 2) [Network1?Sender,Receiver,Message]
8:   proc 2 (ReplayAttack) line 172 "pan_in" (state -) [values: 1!2,-1,1]
8:   proc 2 (ReplayAttack) line 172 "pan_in" (state 3) [Network1!_pid,Receiver,Message]
10:  proc 0 (CommEntity1) line 24 "pan_in" (state -) [values: 2!-1,-1,0]
10:  proc 0 (CommEntity1) line 24 "pan_in" (state 1) [Network2!-(1),-(1),_pid]
11:  proc 0 (CommEntity1) line 25 "pan_in" (state -) [values: 1?2,-1,1]
11:  proc 0 (CommEntity1) line 25 "pan_in" (state 2) [Network1?SecretX,SecretY,Trust]
13:  proc 2 (ReplayAttack) line 173 "pan_in" (state -) [values: 1!2,-1,1]
13:  proc 2 (ReplayAttack) line 173 "pan_in" (state 4) [Network1!_pid,Receiver,Message]
15:  proc 2 (ReplayAttack) line 175 "pan_in" (state 5) [(nempty(Network2))]
17:  proc 2 (ReplayAttack) line 176 "pan_in" (state -) [values: 2?-1,-1,0]
17:  proc 2 (ReplayAttack) line 176 "pan_in" (state 6) [Network2?Sender,Receiver,Message]
19:  proc 2 (ReplayAttack) line 177 "pan_in" (state -) [values: 2!2,-1,0]
19:  proc 2 (ReplayAttack) line 177 "pan_in" (state 7) [Network2!_pid,Receiver,Message]
21:  proc 1 (CommEntity2) line 98 "pan_in" (state -) [values: 2?2,-1,0]
21:  proc 1 (CommEntity2) line 98 "pan_in" (state 2) [Network2?SecretX,SecretY,Trust]
23:  proc 2 (ReplayAttack) line 178 "pan_in" (state -) [values: 2!2,-1,0]
23:  proc 2 (ReplayAttack) line 178 "pan_in" (state 8) [Network2!_pid,Receiver,Message]
25:  proc 2 (ReplayAttack) line 170 "pan_in" (state 1) [(nempty(Network1))]
27:  proc 2 (ReplayAttack) line 171 "pan_in" (state -) [values: 1?2,-1,1]
27:  proc 2 (ReplayAttack) line 171 "pan_in" (state 2) [Network1?Sender,Receiver,Message]
29:  proc 2 (ReplayAttack) line 172 "pan_in" (state -) [values: 1!2,-1,1]
29:  proc 2 (ReplayAttack) line 172 "pan_in" (state 3) [Network1!_pid,Receiver,Message]
31:  proc 1 (CommEntity2) line 102 "pan_in" (state 4) [((Entity_State2==Ready1))]
33:  proc 1 (CommEntity2) line 115 "pan_in" (state 13) [((Entity_State1!=InitContact))]
35:  proc 0 (CommEntity1) line 29 "pan_in" (state 4) [((Entity_State1==Ready1))]
37:  proc 0 (CommEntity1) line 42 "pan_in" (state 13) [((Entity_State2!=InitContact))]
<<<<<START OF CYCLE>>>>>
40:  proc 0 (CommEntity1) line 82 "pan_in" (state 49) [((Entity_State1==Responding))]
      transition failed
spin: trail ends after 40 steps
#processes: 3

```

**Figure 17.4: Property Counterexample for SSL Signed Entities - Replay Attack is recognized and secure communication is stopped**

```

preparing trail, please wait...done
spin: warning, "pan_in", global, 'bit Msg_Sent' variable is never used
spin: warning, "pan_in", global, 'int Msg_Id' variable is never used
spin: warning, "pan_in", proctype DoSAttack, 'int Sender' variable is never used
spin: warning, "pan_in", proctype DoSAttack, 'int Receiver' variable is never used
spin: warning, "pan_in", proctype DoSAttack, 'int Message' variable is never used
spin: couldn't find claim (ignored)
  2:   proc 2 (DoSAttack) line 170 "pan_in" (state -)      [values: 1!2,-1,-1]
  2:   proc 2 (DoSAttack) line 170 "pan_in" (state 1)     [Network1!_pid,-(1),-(1)]
  4:   proc 2 (DoSAttack) line 171 "pan_in" (state -)     [values: 2!2,-1,-1]
  4:   proc 2 (DoSAttack) line 171 "pan_in" (state 2)     [Network2!_pid,-(1),-(1)]
<<<<<START OF CYCLE>>>>>
spin: trail ends after 6 steps
#processes: 3
  6:   proc 2 (DoSAttack) line 169 "pan_in" (state 3)
  6:   proc 1 (CommEntity2) line 95 "pan_in" (state 3)
  6:   proc 0 (CommEntity1) line 22 "pan_in" (state 3)
3 processes created

```

**Figure 17.5: Property Counterexample for SSL Signed Entities – Denial of Service Attack is recognized and secure communication is stopped**

warning: -i or -I work for safety properties only  
warning: for p.o. reduction to be valid the never claim must be stutter-invariant  
(never claims generated from LTL formulae are stutter-invariant)  
(Spin Version 4.0.4 -- 12 April 2003)  
+ Partial Order Reduction

Full statespace search for:  
never-claim +  
assertion violations + (if within scope of claim)  
acceptance cycles + (fairness disabled)  
invalid endstates - (disabled by never-claim)

State-vector 88 byte, depth reached 56, errors: 0

530 states, stored (1060 visited)

1028 states, matched

2088 transitions (= visited+matched)

32 atomic steps

hash conflicts: 0 (resolved)

(max size  $2^{19}$  states)

2.622 memory usage (Mbyte)

unreached in proctype CommEntity1  
line 47, state 21, "Entity\_State1 = Ready1"  
line 73, state 47, "-end-"  
(2 of 47 states)

unreached in proctype CommEntity2  
line 107, state 21, "Entity\_State2 = Ready1"  
line 134, state 47, "-end-"  
(2 of 47 states)

**Figure 17.6: Property verifies for SSL Unsigned Entities when no attack is present – No Attack was recognized**

warning: -i or -I work for safety properties only  
warning: for p.o. reduction to be valid the never claim must be stutter-invariant  
(never claims generated from LTL formulae are stutter-invariant)  
(Spin Version 4.0.4 -- 12 April 2003)  
+ Partial Order Reduction

Full statespace search for:

never-claim +  
assertion violations + (if within scope of claim)  
acceptance cycles + (fairness disabled)  
invalid endstates - (disabled by never-claim)

State-vector 104 byte, depth reached 70, errors: 0

4699 states, stored (9398 visited)  
14469 states, matched  
23867 transitions (= visited+matched)  
2760 atomic steps  
hash conflicts: 53 (resolved)  
(max size 2<sup>19</sup> states)

3.134 memory usage (Mbyte)

unreached in proctype CommEntity1

line 47, state 21, "Entity\_State1 = Ready1"  
line 73, state 47, "-end-"  
(2 of 47 states)

unreached in proctype CommEntity2

line 107, state 21, "Entity\_State2 = Ready1"  
line 134, state 47, "-end-"  
(2 of 47 states)

unreached in proctype ManInMiddleAttack

line 158, state 14, "-end-"  
(1 of 14 states)

**Figure 17.7: Property verifies for SSL Unsigned Entities when Man in the Middle is present – Man in the Middle Attack was not recognized**

```

preparing trail, please wait...done
spin: warning, "pan_in", global, 'bit Msg_Sent' variable is never used
spin: warning, "pan_in", global, 'int Msg_Id' variable is never used
spin: couldn't find claim (ignored)
2:   proc 1 (CommEntity2) line 81 "pan_in" (state 1)   [((Entity_State2==Ready1))]
4:   proc 1 (CommEntity2) line 94 "pan_in" (state 10)  [((Entity_State2!=InitContact))]
5:   proc 1 (CommEntity2) line 95 "pan_in" (state -)   [values: 1!-1,-1,1]
5:   proc 1 (CommEntity2) line 95 "pan_in" (state 11)  [Network1!-(1),-(1),_pid] <merge 0
now @12>
5:   proc 1 (CommEntity2) line 96 "pan_in" (state 12)  [Entity_State2 = InitContact]
<merge 44 now @44>
7:   proc 2 (ReplayAttack) line 143 "pan_in" (state 1) [(nempty(Network1))]
9:   proc 2 (ReplayAttack) line 144 "pan_in" (state -) [values: 1?-1,-1,1]
9:   proc 2 (ReplayAttack) line 144 "pan_in" (state 2) [Network1?Sender,Receiver,Message]
11:  proc 2 (ReplayAttack) line 145 "pan_in" (state -) [values: 1!2,-1,1]
11:  proc 2 (ReplayAttack) line 145 "pan_in" (state 3) [Network1!_pid,Receiver,Message]
13:  proc 1 (CommEntity2) line 101 "pan_in" (state 16) [((Entity_State2==InitContact))]
15:  proc 0 (CommEntity1) line 21 "pan_in" (state 1)   [((Entity_State1==Ready1))]
17:  proc 0 (CommEntity1) line 26 "pan_in" (state -)   [values: 1?2,-1,1]
17:  proc 0 (CommEntity1) line 26 "pan_in" (state 2)   [Network1?SecretX,SecretY,Msg_Received1]
18:  proc 0 (CommEntity1) line 31 "pan_in" (state 6)   [(((SecretX!=-1))|(SecretY!=-1)))]
<merge 0 now @7>
18:  proc 0 (CommEntity1) line 32 "pan_in" (state 7)   [Entity_State1 = Ready1] <merge 44
now @44>
20:  proc 2 (ReplayAttack) line 146 "pan_in" (state -) [values: 1!2,-1,1]
20:  proc 2 (ReplayAttack) line 146 "pan_in" (state 4) [Network1!_pid,Receiver,Message]
22:  proc 0 (CommEntity1) line 21 "pan_in" (state 1)   [((Entity_State1==Ready1))]
24:  proc 0 (CommEntity1) line 26 "pan_in" (state -)   [values: 1?2,-1,1]
24:  proc 0 (CommEntity1) line 26 "pan_in" (state 2)   [Network1?SecretX,SecretY,Msg_Received1]
25:  proc 0 (CommEntity1) line 31 "pan_in" (state 6)   [(((SecretX!=-1))|(SecretY!=-1)))]
<merge 0 now @7>
25:  proc 0 (CommEntity1) line 32 "pan_in" (state 7)   [Entity_State1 = Ready1] <merge 44
now @44>
27:  proc 0 (CommEntity1) line 21 "pan_in" (state 1)   [((Entity_State1==Ready1))]
<<<<<START OF CYCLE>>>>>
spin: trail ends after 30 steps
#processes: 3
30:  proc 2 (ReplayAttack) line 142 "pan_in" (state 9)
30:  proc 1 (CommEntity2) line 102 "pan_in" (state 17)
30:  proc 0 (CommEntity1) line 22 "pan_in" (state 15)
3 processes created

```

**Figure 17.8: Property Counterexample for SSL Unsigned Entities – Replay Attack is recognized and secure communication is stopped**

```

preparing trail, please wait...done
spin: warning, "pan_in", global, 'bit Msg_Sent' variable is never used
spin: warning, "pan_in", global, 'int Msg_Id' variable is never used
spin: warning, "pan_in", proctype DoSAttack, 'int Sender' variable is never used
spin: warning, "pan_in", proctype DoSAttack, 'int Receiver' variable is never used
spin: warning, "pan_in", proctype DoSAttack, 'int Message' variable is never used
spin: couldn't find claim (ignored)
2:   proc 2 (DoSAttack) line 143 "pan_in" (state -)      [values: 1!2,-1,-1]
2:   proc 2 (DoSAttack) line 143 "pan_in" (state 1)    [Network1!_pid,-(1),-(1)]
4:   proc 2 (DoSAttack) line 144 "pan_in" (state -)    [values: 2!2,-1,-1]
4:   proc 2 (DoSAttack) line 144 "pan_in" (state 2)    [Network2!_pid,-(1),-(1)]
6:   proc 1 (CommEntity2) line 81 "pan_in" (state 1)   [((Entity_State2==Ready1))]
8:   proc 1 (CommEntity2) line 94 "pan_in" (state 10)  [((Entity_State2!=InitContact))]
10:  proc 0 (CommEntity1) line 21 "pan_in" (state 1)   [((Entity_State1==Ready1))]
12:  proc 0 (CommEntity1) line 34 "pan_in" (state 10)  [((Entity_State2!=InitContact))]
<<<<<START OF CYCLE>>>>>
15:  proc 0 (CommEntity1) line 69 "pan_in" (state 41)  [((Entity_State1==Responding))]
      transition failed
spin: trail ends after 15 steps
#processes: 3
15:  proc 2 (DoSAttack) line 142 "pan_in" (state 3)
15:  proc 1 (CommEntity2) line 95 "pan_in" (state 11)
15:  proc 0 (CommEntity1) line 69 "pan_in" (state 41)
3 processes created

```

**Figure 17.9: Property Counterexample for SSL Unsigned Entities – Denial of Service Attack is recognized and secure communication is stopped**

warning: -i or -I work for safety properties only  
warning: for p.o. reduction to be valid the never claim must be stutter-invariant  
(never claims generated from LTL formulae are stutter-invariant)  
(Spin Version 4.0.4 -- 12 April 2003)  
+ Partial Order Reduction

Full statespace search for:

never-claim +  
assertion violations + (if within scope of claim)  
acceptance cycles + (fairness disabled)  
invalid endstates - (disabled by never-claim)

State-vector 48 byte, depth reached 64, **errors: 0**

443 states, stored (886 visited)

846 states, matched

1732 transitions (= visited+matched)

32 atomic steps

hash conflicts: 0 (resolved)

(max size  $2^{19}$  states)

2.622 memory usage (Mbyte)

unreached in proctype CommEntity1

line 43, state 20, "Entity\_State1 = Ready1"

line 67, state 43, "-end-"

(2 of 43 states)

unreached in proctype CommEntity2

line 97, state 20, "Entity\_State2 = Ready1"

line 121, state 43, "-end-"

(2 of 43 states)

**Figure 17.10: Property verifies for Non-SSL Client Server Entities when no attack is present – No Attack was recognized**

warning: -i or -I work for safety properties only  
warning: for p.o. reduction to be valid the never claim must be stutter-invariant  
(never claims generated from LTL formulae are stutter-invariant)  
(Spin Version 4.0.4 -- 12 April 2003)  
+ Partial Order Reduction

Full statespace search for:

never-claim +  
assertion violations + (if within scope of claim)  
acceptance cycles + (fairness disabled)  
invalid endstates - (disabled by never-claim)

State-vector 56 byte, depth reached 90, errors: 0

3907 states, stored (7814 visited)

13935 states, matched

21749 transitions (= visited+matched)

2932 atomic steps

hash conflicts: 9 (resolved)

(max size  $2^{19}$  states)

2.827 memory usage (Mbyte)

unreached in proctype CommEntity1

line 43, state 20, "Entity\_State1 = Ready1"

line 67, state 43, "-end-"

(2 of 43 states)

unreached in proctype CommEntity2

line 97, state 20, "Entity\_State2 = Ready1"

line 121, state 43, "-end-"

(2 of 43 states)

unreached in proctype ManInMiddleAttack

line 144, state 14, "-end-"

(1 of 14 states)

**Figure 17.11: Property verifies for Non-SSL Client Server Entities when Man in the Middle is present – Man in the Middle Attack was not recognized**

```

preparing trail, please wait...done
spin: warning, "pan_in", global, 'bit Msg_Sent' variable is never used
spin: warning, "pan_in", global, 'int Msg_Id' variable is never used
spin: couldn't find claim (ignored)
2:   proc 1 (CommEntity2) line 72 "pan_in" (state 1)   [((Entity_State2==Ready1))]
4:   proc 1 (CommEntity2) line 84 "pan_in" (state 9)   [((Entity_State1!=InitContact))]
5:   proc 1 (CommEntity2) line 85 "pan_in" (state -)   [values: 1!2]
5:   proc 1 (CommEntity2) line 85 "pan_in" (state 10) [Network1!2]   <merge 0 now
@11>
5:   proc 1 (CommEntity2) line 86 "pan_in" (state 11) [Entity_State2 = InitContact]
    <merge 40 now @40>
7:   proc 2 (ReplayAttack) line 128 "pan_in" (state 1) [(nempty(Network1))]
9:   proc 2 (ReplayAttack) line 129 "pan_in" (state -) [values: 1?2]
9:   proc 2 (ReplayAttack) line 129 "pan_in" (state 2) [Network1?Message]
11:  proc 2 (ReplayAttack) line 130 "pan_in" (state -) [values: 1!2]
11:  proc 2 (ReplayAttack) line 130 "pan_in" (state 3) [Network1!Message]
13:  proc 0 (CommEntity1) line 18 "pan_in" (state 1)   [((Entity_State1==Ready1))]
15:  proc 0 (CommEntity1) line 23 "pan_in" (state -)   [values: 1?2]
15:  proc 0 (CommEntity1) line 23 "pan_in" (state 2)   [Network1?Msg_Received1] 40:
    proc 2 (ReplayAttack) line 173 "pan_in" (state 4)
40:  proc 1 (CommEntity2) line 116 "pan_in" (state 14)
40:  proc 0 (CommEntity1) line 82 "pan_in" (state 49)
3 processes created

```

**Figure 17.12: Property Counterexample for Non-SSL Client Server Entities -  
Replay Attack is recognized and communication is stopped**

```

preparing trail, please wait...done
spin: warning, "pan_in", global, 'bit Msg_Sent' variable is never used
spin: warning, "pan_in", global, 'int Msg_Id' variable is never used
spin: couldn't find claim (ignored)
2:   proc 2 (DoSAttack) line 126 "pan_in" (state -)      [values: 1!-1]
2:   proc 2 (DoSAttack) line 126 "pan_in" (state 1)     [Network1!-(1)]
4:   proc 2 (DoSAttack) line 127 "pan_in" (state -)     [values: 2!-1]
4:   proc 2 (DoSAttack) line 127 "pan_in" (state 2)     [Network2!-(1)]
6:   proc 1 (CommEntity2) line 72 "pan_in" (state 1)    [((Entity_State2==Ready1))]
8:   proc 1 (CommEntity2) line 84 "pan_in" (state 9)    [((Entity_State1!=InitContact))]
10:  proc 0 (CommEntity1) line 18 "pan_in" (state 1)    [((Entity_State1==Ready1))]
12:  proc 0 (CommEntity1) line 30 "pan_in" (state 9)    [((Entity_State2!=InitContact))]
<<<<<START OF CYCLE>>>>>
15:  proc 0 (CommEntity1) line 64 "pan_in" (state 38)  [((Entity_State1==Responding))]
      transition failed
spin: trail ends after 15 steps
#processes: 3
15:  proc 2 (DoSAttack) line 125 "pan_in" (state 3)
15:  proc 1 (CommEntity2) line 85 "pan_in" (state 10)
15:  proc 0 (CommEntity1) line 64 "pan_in" (state 38)
3 processes created

```

**Figure 17.13: Property Counterexample for Non-SSL Client Server Entities – Denial of Service Attack is recognized and communication is stopped**