Association for Information Systems
AIS Electronic Library (AISeL)

AMCIS 2005 Proceedings

Americas Conference on Information Systems (AMCIS)

2005

Dependability Auditing with Model Checking

Bonnie Anderson
Brigham Young University - Utah, bonnie_anderson@byu.edu

James Hansen
Brigham Young University - Utah, jvh@email.byu.edu

Paul Benjamin Lowry
Brigham Young University - Utah, paul.lowry.phd@gmail.com

Dependability Auditing with Model Checking

Bonnie Anderson
Brigham Young University
Bonnie_Anderson@BYU.edu

James Hansen
Brigham Young University
jvh@email.BYU.edu

Paul Benjamin Lowry
Brigham Young University
Paul.Lowry@BYU.Edu

ABSTRACT

Model checking offers a methodology for determining whether a model satisfies a list of correctness requirements. We propose a theory of dependability auditing with model checking based on four principles: (1) The modeling process should be partitioned into computational components and behavioral components as an aid to system understanding; (2) The complex system will be abstracted to create a model; (3) A language must be available that can represent and evaluate states and processes that evolve over time; (4) Given an adequate model and temporal specifications, a model checker can verify whether or not the input model is a model of that specification: the specification will not fail in the model. We demonstrate this theoretical framework with Web Services and electronic contracting.

Keywords

Model checking, e-processes, web services, electronic contracting, dependability auditing

INTRODUCTION

Systems are generally thought to be correct if they satisfy their design requirements. However, if a system being designed controls a process such as e-business transactions involving concurrency and distributed processes, it is virtually impossible for a designer or system auditor to guarantee that the design requirements are satisfied under all conditions. It is insufficient to simply demonstrate that a system can meet its requirements—the important test is to show that a system *cannot fail* to meet its requirements (Holzmann 2004).

To address this imperative, model checking offers a robust methodology for determining whether a model satisfies a list of correctness requirements. These correctness requirements are conventionally specified as *safety* and *liveness* properties translated into temporal logics or regular expressions. Given today's interest in developing reliable agent-based systems and e-business systems, model checking offers a robust method of evaluating the complexities that can occur in such systems due to their distributed and asynchronous characteristics (Bordini et al. 2004).

Our contribution to the topic of model checking is a theoretical framework suitable for dependability auditing of complex information systems. We demonstrate this theoretical framework with Web Services (WS) and electronic contracting.

BACKGROUND

Recent studies are discovering that e-process managers, developers, as well as system auditors require increasingly robust tools to assure designers and users that e-business systems are secure and reliable. Heintze et al. (1996) used a model checker to examine the non-security characteristics of e-business systems to verify the money and goods atomicity properties of two commercial e-business payment systems: Digicash© and NetBill©. Failures are found, and counterexamples are generated to illustrate a set of actions leading to a state where money atomicity was not realized.

Wang et al. (2001) provide evidence that model checking plays a valuable role in designing e-business systems and is an effective way of evaluating and auditing existing e-business systems. They use a ticket sales application to demonstrate the capabilities of two model checkers. This application abstracts fundamental characteristics of many e-business systems, including distributed processing, parallelism, concurrency, communication uncertainties, and continuous operations. Ray and Ray (2000) extend the basic structures examined by Heintze et al. who assume that neither the NetBill server nor the communication links to the server ever fail. They provide a more comprehensive treatment of system and communication

failures than addressed by Heintze et al. by allowing communication links among customers, merchants, or trusted third parties to fail arbitrarily. Anderson et al. (2005) use the Ray and Ray digital trading model checking to present an accessible explication of the discovery and correction of errors via model checking methods.

We contribute to this research by proposing a theoretical framework for dependability auditing of e-processing systems with model checking. We provide context to our framework and its methods by applying them to two complex and familiar e-processing systems: WS and electronic contracting.

THEORETICAL FRAMEWORK

Establishing robust dependability auditing in distributed transaction environments is of growing concern for managers and auditors. The fundamental notion is that certain important properties associated with e-processing systems are verifiable by automated reasoning through computation. Although it is unusual in current practice to thoroughly test the dependability of applications before their delivery, experts have argued that such changes need to be implemented. (Reimenscheider et al. 2004)

What is the nature of dependability in this context? Holzmann (2004) observes that it is conventional to categorize dependability requirements into *liveness* and *safety* properties, which are translated into temporal logics or regular expressions. A *liveness* property is a characteristic of a system that must hold true for every possible execution of a program. *Liveness* expressions are claims that a requirement will eventually be satisfied, usually relating to desirable features of a system like correct termination, occurrence, responsiveness, and precedence.

A *safety* property is a statement that claims that something will never happen. Such properties generally pertain to undesirable actions or outcomes: deadlock, invariance, absence of undesired states, etc. It should be noted that not every interesting correctness requirement can be neatly classified as a liveness property or a safety property. It has been postulated, however, that any correctness property can be represented as the intersection of a safety property and a liveness property (Holzmann, 2003).

Our dependability-auditing theory is comprised of three constructs: *modeling, specification,* and *verification.* We present and discuss key principles supporting these constructs in the remainder of this section with the objective of providing constructive evidence for each principle.

Principle 1. Models suitable for model checking must define both a computational component and a behavioral component.

Schnoebelen (2002) posited that a computational component is a necessary representation of the system to be examined, in that it describes possible configurations and the potential interactions between them. Since the state of a system may change over time, a behavioral component is essential to describing the dynamics of the computational component.

A computational component can be formally represented as a Kripke structure (Clarke et al. 1999) $M(S, S_o, R, L)$ where

1. S denotes a finite set of states
2. $S_o \subseteq S$ is the set of initial states
3. R = S x S is a transition relation such that for every state $s \in S,$ there is a state $s' \in S$ with $(s, s') \in R$
4. There is a function $L : S \rightarrow 2^{AP}$, which labels each state with the set of atomic propositions true in that state. Atomic propositions are the simplest propositions that can be evaluated as true or false.

A Kripke structure defines a directed graph with nodes *S* that represent the possible systems' states, edges that represent the state transitions *R*, and the paths that are possible system executions. The benefit of structuring models in this manner is that graph theoretic algorithms designed for optimizing search can be applied.

Behaviors tell us how a model evolves from one state to another over time. They are typically defined by finite state *automatons*, which are machines that evolve from one state to another under the action of transitions. An *automaton* is a tuple, (S, s_0, L, T, F) ,where S is a finite set of states, s_0 is a unique initial state $s_0 \in S$, L is a finite set of labels, T is a

set of transitions $T \subseteq (SxLxS)$, and F is a set of final states $F \subseteq S$ (Bordini et al. 2004). A behavior is then defined as an ordered set of transitions $\{(s_0, l_0, s_1), (s_1, l_1, s_2), (s_2, l_2, s_3), ...\}$ such that $(s_i, l_i, s_{i+1}) \in T$ (Holzmann 2004).

Constructing the computational component and the behavioral component does not guarantee a good model; however, when combined with model abstraction it can create an appropriate framework for the evaluation process (Schnoebelen 2002).

Principle 2. Complex systems can often be effectively represented by methods of model abstraction.

As discussed by Clarke et al. (1999) the key to modeling a complex system is the principle of *abstraction*, which retains the essential elements of a system while avoiding the state explosion problem. Given a system S and its model (abstraction) $\alpha(S)$ with possible respective executions $E(S)$ and $E(\alpha(S))$, the objective is that any correctness requirement proven for $E(\alpha(S))$ also holds for $E(S)$. A useful approach to accomplishing this objective is *data abstraction*.

Data abstraction is based on the precept that systems often exhibit straightforward relationships among the data values in the system. Suppose that a system S contains variables that range over a set of values V. To construct $\alpha(S)$ an abstract domain V' is chosen and a mapping *g* from V to V' is defined. This generates a new Kripke structure $M(S, S_o, R, L)$, which is identical to that of S, except that L labels each state with a set of abstract atomic propositions from AP. The structure of M can then be reduced to M_r as follows:

1. $S_r = \{L(s) \mid s \in S\}$, meaning that the set of states in the reduced structure is the set of all labelings of the states of M.
2. $s_r \in S_0^r$ if and only if there exists s such that $s_r = L(s)$ and $s \in S_0$.
3. $AP_r = AP.$
4. Each s_r is simply a set of atomic propositions; thus, $L_r(s_r) = s_r$.
5. $R_r(s_r, s_r')$ if and only if there are s and s' such that $s_r = L(s), s_r' = L(s'),$ and $R(s, s').$

We now claim that M_r simulates M.

Proof:

$H = \{(s, s_r) \mid s = L(s)\}$ can be used as a simulation relation. Therefore whatever properties can be verified in M_r will hold for M. □

Clarke et al. affirm that by using this strategy it is possible to determine whether formulas over AP are true in M, and that AP can be chosen in such a way that it is possible to express the properties of M that need to be checked.

Principle 3. To enable verification, it is necessary to specify the properties a model must satisfy in a language that describes how the behavior of the system evolves over time.

Specification is the description of the desired system properties. This is a critical process in model checking because a system cannot be proven correct in any absolute sense. For example, a universally desired e-process property is that no agent will wait indefinitely for response to a request for information from another agent (Solaiman et al. 2003). Other common properties include temporal requirements such as ensuring that no payment is made until goods are received or that a price never exceeds specified limits.

Linear temporal logic (LTL) is the dominant formalism in distributed system verification (Holzmann 2004). LTL utilizes operators that express temporal modalities such as *eventually* and *henceforth*. These modalities can be applied in a straightforward manner to the description of computations by a transition system. As an example, suppose that $\sigma = s_0, s_1, s_2 ...$ is a computation by a model, and that λ is a formula to be evaluated. At any point in the process one can ask not only whether λ is true in s_i, but also whether λ is true in at least one of the states $s_i, s_{i+1}, s_{i+2}, ...$ that the

computation will visit from point I onward in λ. If this is true, the formula F λ is true at *i* in σ otherwise F λ is false. In addition, the formula G λ is true at point I in σ if λ is true in s_i and in each and every one of the states will visit after s_i in σ. In other words, the temporal logic operators F and G correspond to the modalities *eventually* and *henceforth.* Finally, there are operators X and $\cup$ that correspond to the modalities *next* and *until*. The formula X λ is true at point *i* in σ if it is true at point *i+1* in σ. The formula $\theta \cup \lambda$ is true at point i in σ if λ is true at some point in σ after *i* and θ is true from point *i* until λ becomes true.

A simple example will suggest the robustness added by LTL. Consider the expression that p implies q $(p \rightarrow q)$. Here no temporal operators exist, so this property must hold for every execution of the system. From the above definitions, however, an intended temporal property that *will always hold that p will become true at some time and the next operation will ensure that q is true* would be expressed as $G(p \rightarrow> X(F(q)) \; and \; F(p)$, which is markedly different from $(p \rightarrow q)$ (Holzmann 2004). Specification facilitates expression of required system properties. These specifications are made robust by the use of temporal logics.

Principle 4. Given a Kripke structure M representing a distributed system and a temporal logic formula λ expressing a required property, verification determines if M is a model of λ ($M \models \lambda$).

We may think of a specification λ as expressing a property that can be interpreted as true or false. If the execution of M results in an interpretation of λ as true, we say that M is a model of λ or that M satisfies λ ($M \models \lambda$).

Verification is then the mechanism by which it is determined whether a system model satisfies its specifications; or verification is a check to see that the model does what we want it to do. Model checkers either confirm that given specifications hold or report that they are unsatisfied, which has been proven by Clarke et al. Given sufficient resources, model-checking procedures will always terminate with a *yes/no* answer (Holzmann 2004).

A powerful feature of model checker verification is that if the desired properties are unsatisfied a *counterexample* is provided, which is a trace of the processes that lead to specification failure. Counterexample traces are critical to discovery of subtle errors in complex distributed systems. We denote E as the set of all possible runs of a system. The verification process demonstrates that either E does not contain any runs that violates λ or provides positive evidence that at least one such run exists—in which instance it becomes a counterexample.

Although model checking is intended to verify that necessary properties are satisfied in a given system design, failures to satisfy specifications can also result from incorrect modeling of a system or from an incorrect specification. Counterexample traces can be useful in identifying and resolving these issues. We now illustrate our theoretical framework of model checking using WS and electronic contracts. The model checker SPIN (Holzmann 2004) is used for these studies.

WEB SERVICES

Our first study of model checking with our theoretical framework involves WS. Schlingloff et al. (2003) define WS as software systems designed to support interaction over a network. They also develop the structure and behaviors that are fundamental to the WS application of this section. WS are closely related to agent programming paradigms that have the objective of standardizing various aspects of distributed processing and Internet communication. The motivation for WS is the straightforward flexibility of architecture designed to facilitate distributed computation via inter-agent communication. Such capability is implemented via remote procedure calls between distributed objects. Communication in such settings consists fundamentally of passing values to remote procedures with the receipt of messages containing the results.

In WS, a registry can receive requests from customers over a known global channel. When a registry accepts a request for service, it can assign that request to an agent and can then provide a private channel name to the customer that can be used to communicate with the agent. Further interactions are between the agent and the customer, who communicate across the private channel without further intermediation by the registry. Once the transaction is complete, the agent returns the identifier for the private channel to the registry and awaits the next transaction.

A customer sending a request to a registry may attach the name of the channel where it will listen for responses from the registry or the registry's agent. Eventually, when the customer transaction is complete, the registry's agent will return the available private channel so that the registry can add it to its pool of available channels.

Consider a WS that is available to a potential customer on the Internet. Such a system will be characterized by a dynamically changing number of active processes. Following the above description, a WS agent, whom we label *WSAgent*, is able to receive requests from potential customers over the Internet. When the WSAgent receives a request it assigns that request to a customer agent, which we label *WSCustAgent*, and provides a private channel name to the customer. This channel is then used to communicate with the WSCustAgent. The rest of the transaction takes place between these two parties without further action by the WSAgent. Once the transaction is completed, the WSCustAgent returns the identifier for the private channel to the WSAgent.

A customer sending a request to the WSAgent attaches the name of the channel where it will listen for responses from the WSAgent or the WSCustAgent. If the channel set is empty, which could occur if demand is heavy, then the WSAgent has no choice but to deny the request at that time. If a channel is available, an agent interchange is initiated and the name of the private channel is communicated to the customer, along with the channel through which the customer can be reached. The WSAgent then returns waiting for new customer agent requests.

When the customer transaction is complete, the WSAgent will return the freed private channel to the set of free channels. This can be modeled so that the WSAgent randomly decides to either grant or deny a request or to inform the customer that the request is on hold. When customer requests are granted, the WSAgent moves to a state where it expects the WSCustAgent to eventually return notification that the transaction is now complete. The WSCustAgent then advises the WSAgent that the private channel is now available.

We now describe our actual model checking results of this design. The first dependability audit that was executed on the WS model yielded the results reproduced in Figure 1. Of immediate interest is the first line indicating that there is an *invalid end state found at depth 39*. This indicates that a model specification has failed and that a counterexample is available to trace the actions leading to the failure. We also see confirmation of one error and that the model has reached depth 40.

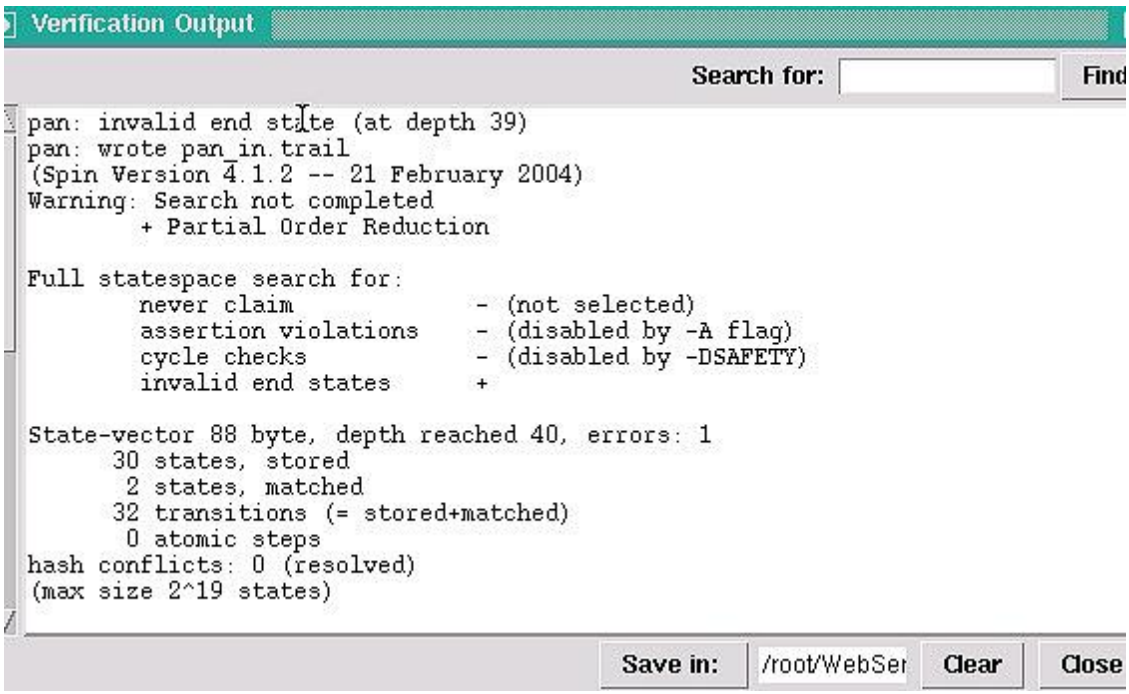

Figure 1. WS Verification

Given this, the model checker was used to generate the counterexample found in Figure 2. Observe that actions terminated immediately after the WSAgent action *server!return, agent*—meaning that the WSAgent sent a message to the WSCustAgent. Inspection establishes that the correct action should have been *server?return,agent*—meaning that the WSAgent should have requested that the WSCustAgent send a message advising when the current customer transactions had been completed.

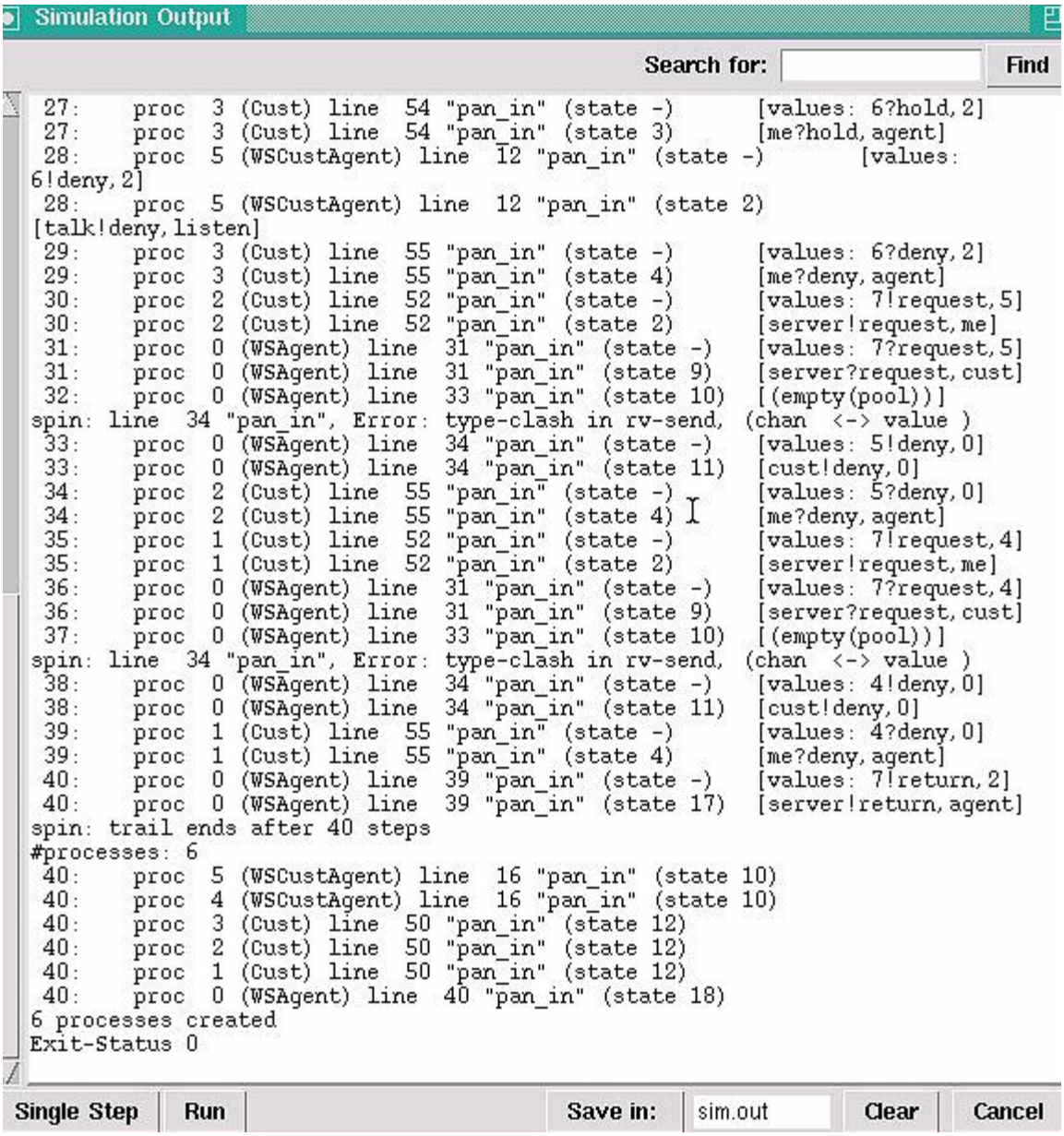

Figure 2. Path of Counterexample for WS

When the above modification is made, an iteration of the verification facility yields output that indicates that there are two unreached states: one in WSCustAgent (line 42) and one in Cust (line 63). Examination of the associated counterexample reveals that there is no instruction to the WSAgent (such as *hold* or *deny*) as to what action to take if no WSCustAgent is available. When these refinements are made, another iteration of the model checker establishes that the WS model satisfies its specific safety requirements, as well as general safety properties.

The first of these general safety properties is freedom from *deadlock*: There are no situations found where further progress in completing a process halts. The second property is *reachability*: Each and every state within the model should be reachable in at least one of its executions. Two such failures were found and resolved, as described above. The third general safety property is the absence of unsolicited responses by agents. No execution was found where this requirement failed. We note that in this process the model checker has investigated a large state-space—including over 4000 states.

Unlike safety properties there are no general liveness properties: such properties are specific to the model being verified. They include such requirements as "price >= 30", "an agent should only wait a fixed period for a response," or "there should always be two channels available for communication between agents." Due to space limitations we do not provide further

details here; but in the actual analysis we did use LTL to verify that two communications channels are always available for both agent types: WSAgent and WSCustAgent.

ELECTRONIC CONTRACTING

Our second study of involves electronic contracting, which application is adapted from Solaiman et al. (2003), who address the complexities of converting a conventional contract written in natural language into an electronic equivalent. The ambiguities that the human text is likely to contain are particularly difficult. To detect and remove these ambiguities, a contract needs to be described in a mathematically precise notation, which can then be subjected to analysis.

A conventional contract is a paper document stipulating that the signatory parties agree to observe the specified agreements. The purpose of a contract is the enforcement of the rights and obligations of the contracting parties. Inconsistencies in the clauses of conventional contracts are common rather than exceptional and human expertise is relied upon to resolve those inconsistencies. Conversely, electronic contracts must avoid inconsistencies; hence, knowing the correctness requirements of an e-contract at design time is critical in the sense that the e-contract be proven correct with respect to a specific set of correctness requirements. Although the correctness requirements of contracts may vary, Solaiman et al. argue that it is possible to provide a list of standard correctness requirements and to generalize them including:

1. An e-contract should begin its execution in a well defined initial state prescribed by the occurrence of a specified event.
2. Each state in the e-contract should be executable in a least one of the execution paths of the contract.
3. An e-contract should never enter a situation in which no further progress is possible.
4. If an e-contract commences its execution with a precondition true then the precondition should remain true for the duration of the contract.
5. E-contracts should not allow unsolicited responses.
6. An e-contract must reach a well-defined termination state.
7. A request for a service will receive a response by the end of a finite period of time.

Requirements 1-5 are *safety* properties. Requirements 6 and 7 are *liveness* properties.

The sources of contract inconsistencies are thought to derive from two factors: (1) internal enterprise policies conflicting with contractual clauses and (2) inconsistencies in the clauses of a contract. The model of interest here focuses on the second source and is motivated by the fact that the principal concern is with the cooperative behavior of business enterprises and not their internal structure. These business interactions can be represented as finite state machines. Use of finite state machines for representing such interactions has been proposed for a number of distributed computing applications.

In the current contract model each of the contracting parties has the privilege and responsibility of verifying that its internal policies do not conflict with the contract clauses. Each enterprise exercises its independence in choosing the roles players that would invoke operations on the contract and provide them with a proper contract role player certificate.

The e-contract contains two processes: one for the Purchaser and one for the Supplier. The Purchaser requests an offer for a product or service from the Supplier and then awaits a response. The Supplier then considers the request and rejects the request or responds with an offer. If the Supplier's response is an offer, then the Supplier awaits a response from the Purchaser. The Purchaser may respond with an acceptance of the offer, a rejection of the offer, or a counteroffer. In the case of acceptance, the Supplier completes the transaction. In the case of rejection, the Supplier can abort the transaction or make another offer, in which case the entire process repeats. In the case of a counteroffer, the supplier can accept, reject, or make another offer itself. If the Supplier accepts the Purchaser's counter offer the transaction is completed. If the Supplier rejects the Purchaser's counter offer, the transaction is aborted. If the Supplier makes a counter offer, the process is repeated. There are controls on the number of times that the counter offer process can be repeated.

We now describe our model checking results of this design. We first checked the general safety properties, which were satisfied. A check was then made of liveness properties 6 and 7 from the previous section. Figure 3 shows that an error was detected l: *invalid end state (at depth 11)*. The path was saved where the error was detected. To trace the point at which the error occurred, the model checker provides the counterexample shown in Figure 4. Here it is observed that following step 10 the Supplier was expected to send an offer to the Purchaser; however, the counterexample does not show this action. Inspection of step 11 reveals that the trail ended after the simulator executed line 33 of the verification model. In particular,

```
31      if
32      :: S2P|Offer(offerValue) -> goto WaitingForResults;
33      :: skip /*Taking into account the possibility that*/
```

34 fi; /*the supplier might not send anything*/

Line 33 denotes that the Supplier might choose not to send the offer to the Purchaser. Figure 4 also shows that the model checker detects problems in lines 60 and 37:

WaitingForOffer:

S2P ? Offer(offerValue) ->

WaitingForResults:

P2S ? Response(responseValue);

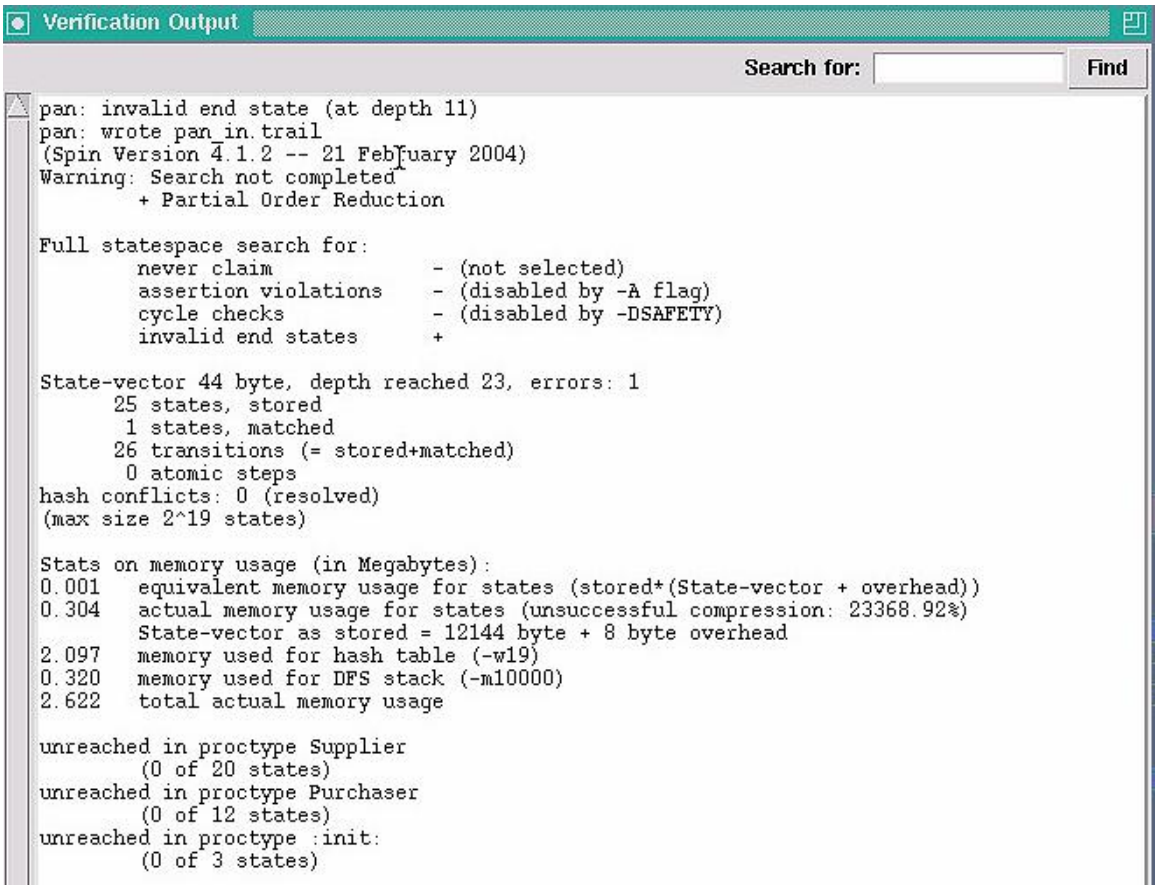

Figure 3. 'Invalid End State'

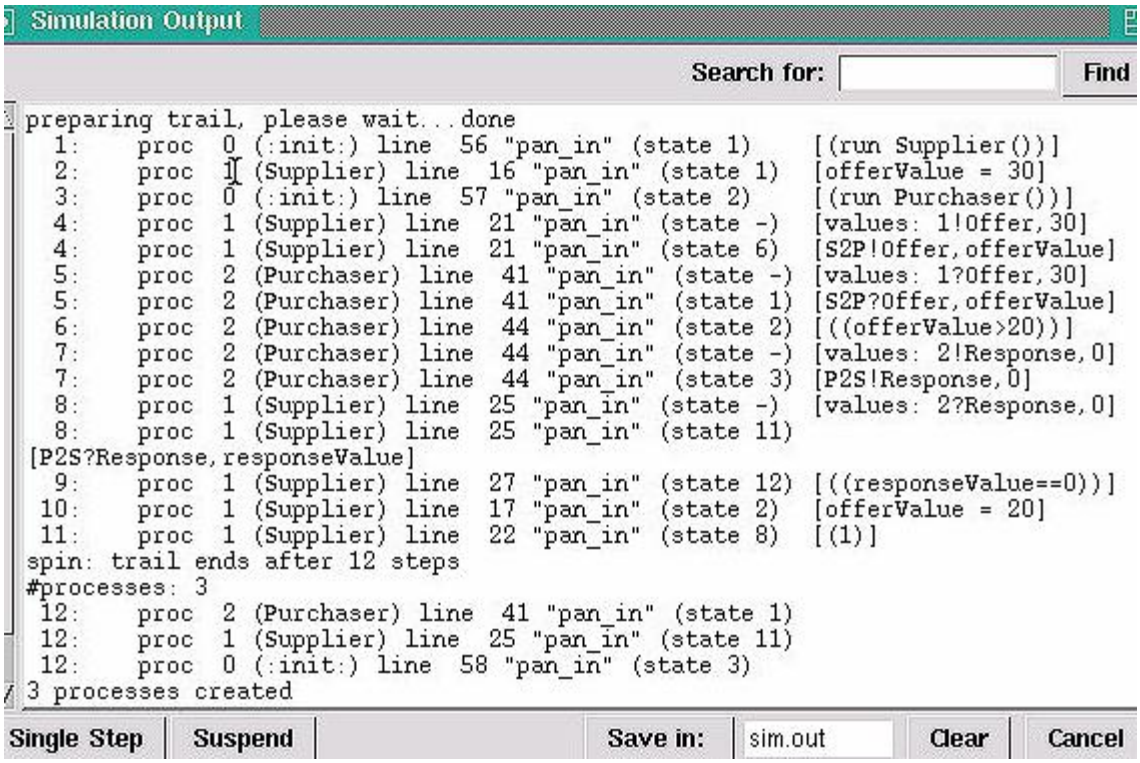

Figure 4. Path of counterexample for electronic contracting

An offerValue was received by the Purchaser process, yet subsequently no responseValue was received by the Supplier process. The result is that the Supplier and Purchaser fall into a deadlock situation. This can be resolved by use of a *timeout* statement, which allows a process to abort and not wait indefinitely for a condition that can no longer become true, such as the one just described. The relevant revision is reproduced below.

```
WaitingForOffer:
if
        ::S2P ? Offer(offerValue)
        ::timeout -> goto end
fi;
```

The modified model is then checked and found to satisfy its general safety requirements. After assuring the correctness of the general safety requirements, the model checker can be used to check for specific safety correctness requirements. In this application we want to ensure that in all executions it is true that "The price offered by the Supplier should not be accepted by the Purchaser if the price exceeds an agreed upon price, P". To check this invariant we can insert an assertion of the form *assertofferValue <=P* at the required check points in the verification model. The model checker can then be set to check for assertions. In the refined application, the model checker affirms the required invariant value.

CONCLUSION

The major goal of model checking is to build and verify the smallest sufficient model to describe the essential elements of a system's design (Holzmann 2004). Although there is no guaranteed formula for achieving this, we propose that our theoretical framework captures the key principles necessary to achieve this goal. To assess the generalizability of this theoretical framework for model checking, we conducted two studies that involved modeling processes with WS and electronic contracting applications.

We also showed that the partitioning of the model into conceptual areas of computation and behavior facilitates the identification and content of specifications to be checked. For example, identification of when and where key behaviors occur, or when concurrent activities are taking place, informs specifications concerning invariants, communication channels,

checking of allowed values, indefinite waits, and the logic of sending and receiving messages. All the non-general specifications used in the examples of this paper, were the result of such analysis.

The key elements of the verification process are (1) that it evaluates all possible traces for a specification (exhaustive), and (2) when a failure is found the trace of transitions leading to that failure is available for inspection. We have shown how this can be used to correct and improve e-process models.

There is a dearth of research in extending model checking to e-processes; yet additional research is indicated if the complexities of the future are to be dealt with effectively. As e-processes become more interconnected and more complex with development of computing grids, the Semantic Web, and pervasive computing, the costs of system malfunction or failure will become such that verification increasingly important—even critical in e-process design and implementation.

REFERENCES

1. Anderson, B.B., Hansen, J., Lowry, P.B., and Summers, S. "Model checking for design and assurance of e-Business processes," *Decision Support Systems (DSS)* (39:3) 2005, pp 333-344.
2. Bordini, R., Fisher, M., Visser, W., and Wooldridge, M. "Model checking rational agents," *IEEE Intelligent Systems* (19:5) 2004, pp 46-52.
3. Clarke, E., Grumberg, O., and Peled, D. *Model Checking* The MIT Press, Cambridge, MA, 1999.
4. Heintze, N., Tygar, J., Wing, J., and Wong, H. "Model checking electronic commerce protocols," 2nd USENIX Workshop in Electronic Commerce, 1996, pp. 147-165.
5. Holzmann, G. *The Spin Model Checker* Addison Wesley, Boston, MA, 2004.
6. Ray, I., and Ray, I. "Failure analysis of an e-commerce protocol using model checking," 2nd International Workshop on Advanced Issues of e-commerce and Web-based Information Systems, Milpitas, CA, 2000, pp. 176-183.
7. Reimenscheider, R., Saidi, H., and Dutertre, B. "Using model checking to assess the dependability of agent-based systems," *IEEE Intelligent Systems* (19:5) 2004, pp 62-70.
8. Schlingloff, H., Martens, A., and Schmidt, K. "Modeling and model checking web services," Elsevier, 2003.
9. Schnoebelen, P. "The complexity of temporal logic model checking," *Advances in Modal Logic* (4) 2002, pp 1-44.
10. Solaiman, E., Molina-Jimenez, C., and Shrivastava, S. "Model checking correctness properties of electronic contracts," *Lecture Notes in Computer Science* (2910) 2003, pp 303-318.
11. Wang, W., Hidvegi, Z., Bailey, A., and Whinston, A. "Model checking--A rigorous and efficient tool for e-commerce internal control and assurance," GBS-DIA-2001-07, Georgia State University, Atlanta, GA.