Automatic Support for Verification of Secure Transactions in Distributed Environment using Symbolic Model Checking

E. Di Sciascio, F. M. Donini, M. Mongiello and G. Piscitelli

Dip. Elettrotecnica ed Elettronica, Politecnico di Bari, Italy

Electronic commerce needs the aid of software tools to check the validity of business processes in order to fully automate the exchange of information through the network.

Symbolic model checking has been used to formally verify specifications of secure transactions in a system for business-to-business applications. The fundamental principles behind symbolic model checking are presented along with techniques used to model mutual exclusion of processes and atomic transactions. The computational resources required to check the example process are presented, and the faults are detected through symbolic verification.

Keywords: e-commerce, model checking, secure transactions

1. Introduction

Electronic commerce is the ability to perform business transactions involving the exchange of goods and services between two or more parties using electronic tools and techniques. In electronic commerce information is conveyed through electronic networks and computer systems and much of the transaction is automated. The kernel of electronic business is the business transaction and issues such as security and trust are the most important challenges in conducting an automated transaction. Electronic commerce applications do not yet provide robust transactions, messaging and data access services typical of contemporary client/service applications (Manchala 00).

In this paper we propose to conduct a complete search for process faults in the field of business transactions using formal methods for verifying safety and liveness properties of secure transactions, in order to improve process integrity. The approach we propose is based on model checking, a powerful formal verification method that determines whether a system model satisfies certain specifications under all circumstances (Emerson 86). Model checking can locate subtle but critical flows that conventional assurance methods such as testing and simulation often miss. Model checkers, on the other hand, are fully automated tools to verify that a system model satisfies certain properties. Automation makes this formal method particularly attractive to business process.

We used a verification tool — Symbolic Model Verifier (SMV) — to formally verify safety properties of business transactions. To accomplish this, we used a model checking method: we created finite-state models of the system of interest and also identified and expressed relevant properties to verify in a properly defined logical language, CTL (Pnueli 77). We discuss the modeling of systems for internet-based business and transaction processes. We also show the formulation of specifications and discuss fundamental issues concerning the atomic transactions. With respect to other modeling techniques, such as UML (Conallen 99) or models based on object-oriented approach (Manchala 00), the approach based on model checking guarantees the automation of the verification

procedure. Our results from the verification are included to show the current capabilities of the modeling techniques and the model checker.

2. Symbolic Model Checking

The power of the method is that it is automatic and fast. Automatic model checkers have been implemented using algorithms for checking the truth of temporal logic specifications on finite-state models (Katoen 99). The model checker helps in finding errors in the design of system by producing a counter example that can be used to correct the model. The first model checkers represented models as graphs, where nodes in the graph represented states of the system and edges denoted possible transitions between states. Models are assumed to be Kripke structures, e.g. triples of the form $M = (S, R, L)$ where S is a set of states, $R \subseteq S \times S$ is a relation which defines legal state transitions (i.e. the transition relation), and L is a function that labels each state with the atomic propositions that hold true on that state (Clarke 86). It is possible to algorithmically check such a model for satisfaction of temporal logic properties. It is also possible to algorithmically generate counterexamples when a model does not satisfy a temporal logic property. Therefore, given a model M and temporal logic properties expressed in an appropriate language, model checking procedures can be automated.

The model checking procedure carried out on graphs was efficient, but since the number of states in a finite-state model grows exponentially with the number of state variables, only small systems ($10^3 - 10^6$ states) could be checked (Clarke 99). In fact, the main challenge in model checking is the state explosion problem. That problem occurs because of the many components that interact in a system or because of the concurrency properties among them. The state explosion problem was addressed by McMillan (1992), who introduced symbolic model checking.

Symbolic model checking is a variation of model checking where sets of states and the transition relation are represented implicitly using Boolean formulas rather than an explicit graph structure (Bryant 86). A set of states may be represented by Boolean formulas that hold true on those states.

Symbolic model checking reduces the state explosion problem. These Boolean formulas are in turn represented and manipulated in a very efficient manner with Ordered Binary Decision Diagram (Bryant 92). Systems with as many as 10^{120} reachable states have been verified using symbolic model verification techniques.

Transitions may be represented by a Relation $R(v, v')$ where v is the vector of current assignments to the states components (atomic propositions) and v' is the vector of next state assignments. An edge exists in the graph if R is true for two state vectors v and v'. If the set of initial states and the transition relation are given for a model, the reachable state space may be found.

3. Computation Tree Logic (CTL)

Properties to be verified are expressed in a propositional, branching, temporal logic named Computation Tree Logic (CTL). We do not define here CTL; for an introduction, see (Clarke 99). Any propositional logic formula is a CTL formula. CTL formulas may also contain path quantifier followed by temporal operators. The path quantifier E specifies some path from the current state while the path quantifier A specifies all paths from the current state. The temporal operators are X, the next-time operator, U, the until operator, and G, the always operator. $X\phi$ specifies the ϕ holds in the next state along the path. $\phi\ U\ \varphi$ specifies that ϕ holds on every state along the path until φ is true. $G\phi$ specifies that ϕ holds on every state along the path.

Given the above information, CTL can be defined by the following statements:

- Every atomic proposition is a CTL formula. An atomic proposition is the formula true or a state variable assignment.
- If ϕ and φ are formulas, then $\neg\phi$ and $\phi \wedge \varphi$ are formulas
- If ϕ and φ are formulas, then $\mathbf{EX}\phi$, $\mathbf{E}(\phi\ \mathbf{U}\ \varphi)$ and $\mathbf{EG}\varphi$ are formulas

The formulas false, $\phi \wedge \varphi$, $\phi \rightarrow \varphi$, $\phi \oplus \varphi$, $\phi \leftrightarrow$, $\mathbf{EF}\phi$, $\mathbf{AX}\phi$, $\mathbf{AG}\phi$, $\mathbf{AF}\phi$, $\mathbf{A}(\phi\ \mathbf{U}\ \varphi)$ can be derived from the fundamental formulas given

above. Fϕ means that ϕ will hold at some future state along the path.

Intuitively, the temporal expression **AG**ϕ means that the property ϕ is true in **A**ll paths, and **G**lobally (every state), while dually, **EF**ϕ means that there **E**xists a path in which ϕ will become true (in some **F**uture state). **X** refers to the ne**X**t state, with the same meaning for **A**, **E**.

The following are some sample CTL formulas with their English explanations.

- $\neg$**EF**$(\phi \wedge \varphi)$: There does not exist a future state on which ϕ and φ are simultaneously true.
- **EFEG**$(\neg\phi)$: Some future state will be on a path in which ϕ is never true.
- **AG**ϕ : ϕ holds on every reachable state.

Symbolic Model Checking Implementation

The CTL model checking method has been implemented in the package SMV (for symbolic model verifier) (McMillan 92). SMV has its own language for defining finite-state concurrent systems (i.e. the transition relation and the set of initial states). Once the transition relation is built, SMV executes the model checking procedures for the CTL properties. The SMV input language has provisions for representing systems which are hierarchical, modular and nondeterministic. SMV also provides counterexamples whenever necessary or possible.

SMV uses Ordered Binary Decision Diagrams (OBDDs) for efficient representation and manipulation of Boolean formulas. OBDDs are directed, acyclic graphs, which are more compact than other currently used representations (Bryant 86). They are canonical *w.r.t.* a fixed variable ordering, so identical functions are isomorphic (useful for recognizing fixpoints). The number of nodes required to represent a function is sensitive to the variable ordering. Any Boolean operation can be applied to OBDDs, and functions may be substituted for variables, which permits the implementation of existential quantification routines (Bryant 92).

SMV has been successfully used to verify electrical and computer hardware including synchronous protocols for distributed multiprocessors (SMV).

4. The Proposed Model

We have developed the finite-state model of an atomic transaction and defined the set of specifications expressed in CTL to define the correct behavior of a system for business-to-consumer and business-to-business applications. The verification has been carried out using the SMV tool, to detect undesired events in the system behavior, otherwise hardly detectable, and to properly refine the model. Fig. 1. a) describes the flow of operation in a generic business-to-consumer transaction; Fig. 1. b) describes interactions for a business-to-business application. In this section we describe the variables, processes and operating procedures that we use to model a business process.

Mutual Exclusion of Concurrent Processes

We model the system using three concurrent processes: *Consumer*, *Producer* and *Server*. The processes share two resources: Buffer and Database, in accordance with a well-known textbook exercise (Huth 99).

Variables Involved in the System Model

The system behavior is described through the set of variables in the Table 1.

Possible values for variables *Buffer* and *Database* are: *full*, *empty or not full*. The information concerning the authenticity of the signature — *Infosign* — can be *valid*, *not valid* or *unknown*. The kind of information, e.g. *TypeInfo*, can be *critical* or *not critical*.

According to the mutual exclusion problem, we model three processes *Producer*, *Consumer* and *Server* that might be in three different states: critical, trying, and non critical; since there are two resources respectively, *Buffer* and *Database* we distinguish such states as *trying1*, *trying2* and *critical1*, *critical2* depending on the resource the processes are waiting for, e.g, *trying1* and *critical1* for the *Buffer* and *trying2* and *critical2* for the *Database*. Moreover we use other five operating procedures to complete

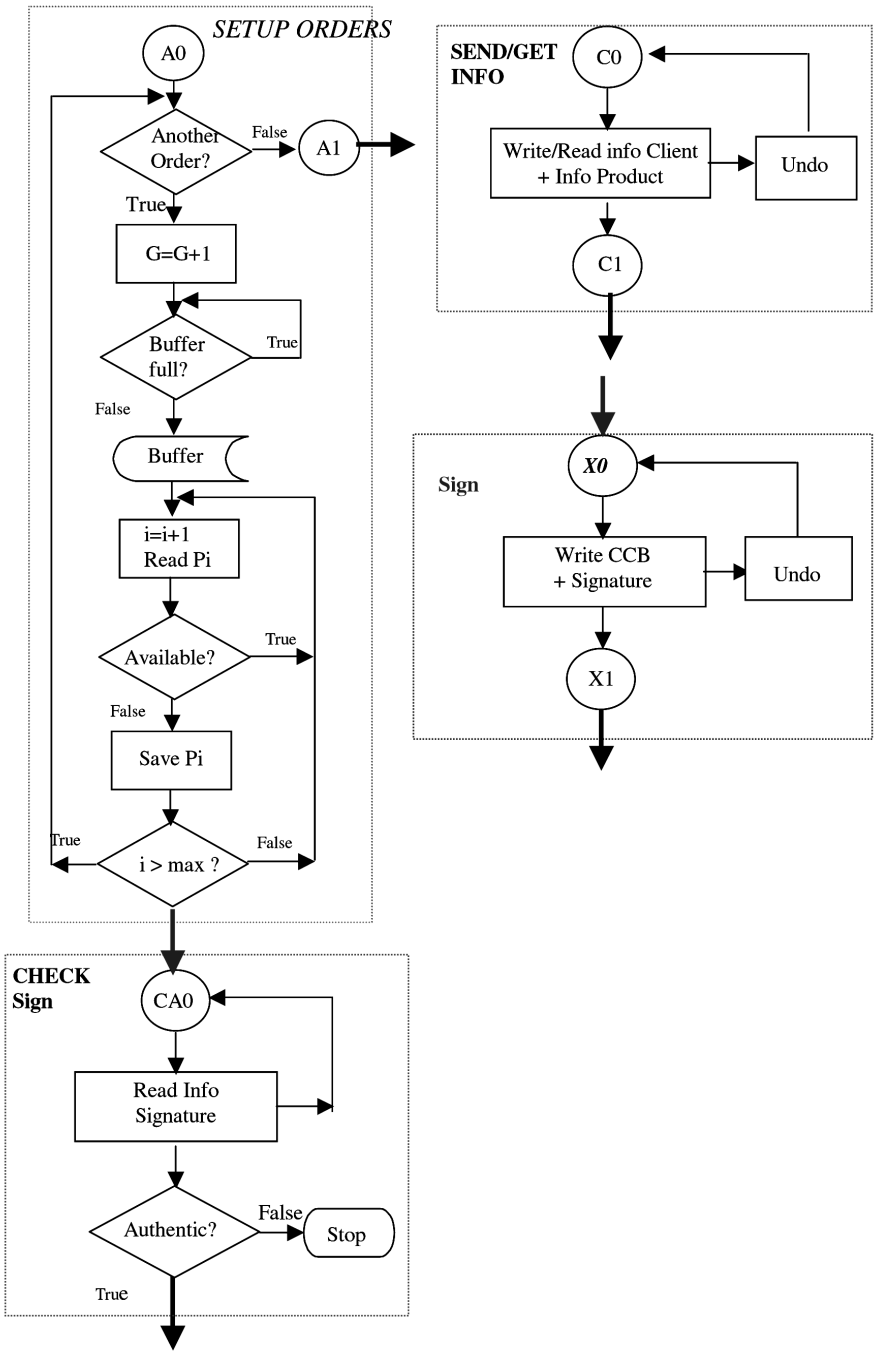

Fig. 1. (a) Model of a Business-to-Consumer operation;

AddSign	true if it is necessary adopt a security mechanism
InfoSign	specify the authenticity of the signature
GetSign	true if it is necessary to check information referring to the validity of the signature
TypeInfo	specify whether the information may be critical
Balance	the amount of bank credit available for the purchaser
Request	true if a request arrives to the system

Table 1. The variables of the system.

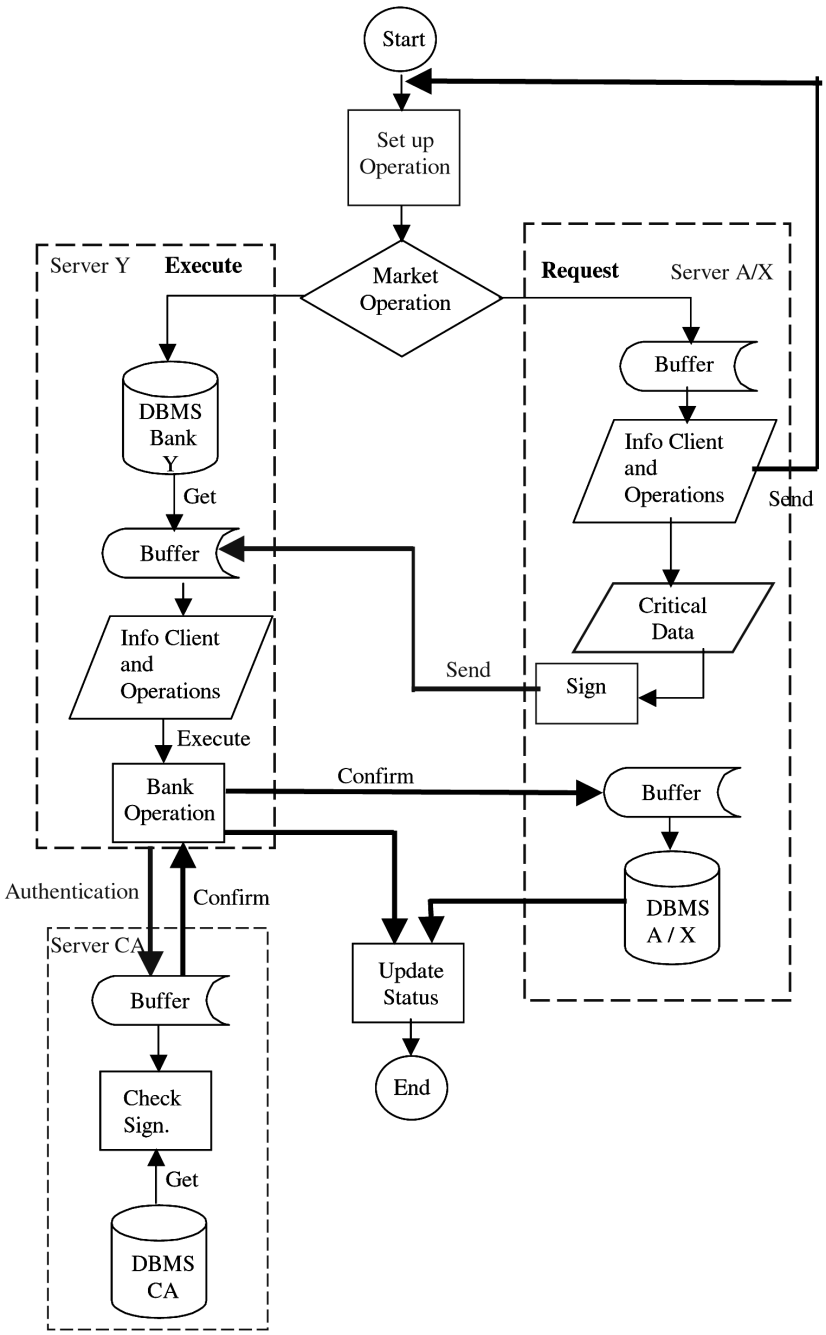

Fig. 1. (b) Model of a Business-to-Business operation.

the transaction: *BankTransfer, CheckSign, GetReq, Transaction* and *SetupReq*. *BankTransfer* performs the payment task between the two actors of the transaction; *CheckSign* checks the authenticity of the signature; *GetReq* extracts from the *Database* the requests that arrive to the *Server*, that should be served by *Transaction*. *Transaction* is the process that executes the transaction, by means of the other processes and serves the request. *SetupReq* sets the information necessary for the execution of a transaction.

The overall model is constructed by asynchronous steps (interleaving) among all the other four processes: *Checksign, Transaction, SetupReq* and *GetReq*. The process *BankTransfer* has a synchronous execution and can be in one of the following states: *success, error, run, idle*.

Atomic Transactions

In Fig. 2 we show the transition diagram that describes how the system moves through its execution states. The first transition occurs as soon as a request arrives to the system. The *Producer* enters its critical region and the request is queued to the *Buffer*. In the next transition the *Consumer* enters the critical region to store the request in the *Database* and the *Server* schedules the request.

The *Transaction* procedure performs two different market operations — *Purchase* and *Sell*. The system can be in two different states depending on the operation. In case of *Purchase* a *Start Setup Request* procedure provides proper information to the supplier according to the request. For a *Sell* operation the diagram describes the execution states of an atomic transaction. The transaction goes in a *run* state after it starts execution. Process *Start Check Sign* checks the digital signature. Once the authen-

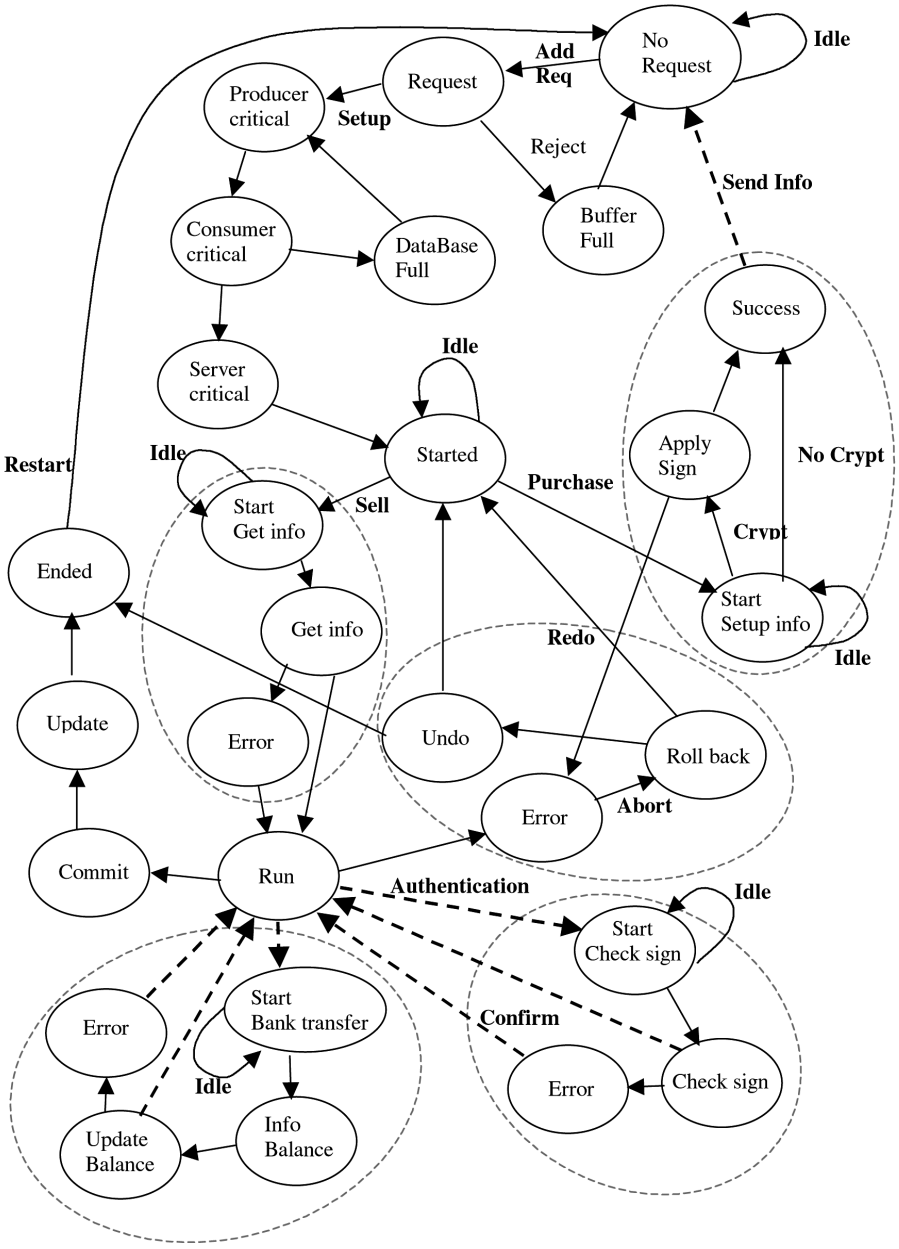

Fig. 2. Business-to-Business transaction model.

tication is obtained, *Start Bank Transfer* can be executed. If the transaction ends successfully, it goes in the *commit* state and then in the *update* state so that all the changes executed by the transaction will not be undone.

The *Transaction* enters the *roll-back* state after the error one if an *error* occurred during the processes execution. Two recovery states are available when the transaction is unsuccessful: *undo* to undo the effects of a transaction and *redo* to specify that some operations must be redone to ensure that all the operations of a committed transaction have been applied successfully.

5. Tested Specifications

In the following we exemplify how the properties to verify have been expressed in CTL.

Mutual Exclusion

A first set of properties to be verified regards mutual exclusion of concurrent processes, as described in (Huth 99).

Access to Resources

The following set of properties ensures that each process tries to access to the shared resource *iff* resources are available: i.e. the *Producer* cannot access to the *Buffer* if it is *full* or the *Server* cannot access to the *Database* if it is *empty*.

For any state if process tries to access a resource then it will eventually be available

1. $\mathbf{AG}((Producer = trying1) \rightarrow \mathbf{AF}((Producer = critical1) \wedge (Buffer = full)))$
2. $\mathbf{AG}((Consumer = trying1) \rightarrow \mathbf{AF}((Consumer = critical1) \wedge (Buffer = empty)))$
3. $\mathbf{AG}((Consumer = trying2) \rightarrow \mathbf{AF}((Consumer = critical2) \wedge (Database = full)))$
4. $\mathbf{AG}((Server = trying2) \rightarrow \mathbf{AF}((Server = critical2) \wedge (Database = empty)))$

Correctness of the States

The following properties ensure that each process reaches the correct states: the *Producer* tries to access to the *Buffer* and not to the *Database*:

5. $\mathbf{EF}(Producer = critical1)$
6. $\mathbf{EF}(Producer = trying1)$
7. $\mathbf{EF}(Server = critical2)$
8. $\mathbf{EF}(Server = trying2)$
9. $\mathbf{EF}(Consumer = critical1 \vee Consumer = critical2)$
10. $\mathbf{EF}(Consumer = trying1 \vee Consumer = trying2)$

For any state if the process does not try to access the wrong resource

11. $\mathbf{AG}\neg\ (Producer = critical2)$
12. $\mathbf{AG}\neg\ (Producer = trying2)$
13. $\mathbf{AG}\neg\ (Server = critical1)$
14. $\mathbf{AG}\neg\ (Server = trying1)$

Properties of Transactions

This set of properties specifies the properties that a transaction must satisfy:

For any state if a request arrives to the system then it will be served in some future state

15. $\mathbf{AG}(\neg\ \mathit{Database}{=}\ \text{empty} \rightarrow \mathbf{EF}(\mathit{Transaction}.\text{state}{=}\text{started}))$

For any state if an error occurs during the transaction then it will eventually be rolled-back

16. $\mathbf{AG}((\mathit{Transaction}.\text{state}{=}\text{error}) \rightarrow \mathbf{AF}(\mathit{Transaction}.\text{state}{=}\text{roll\_back}))$

For any state if the transaction starts then it will be executed in some future state

17. $\mathbf{AG}((\mathit{Transaction}.\text{state}{=}\text{started}) \rightarrow \mathbf{EF}(\mathit{Transaction}.\text{state}{=}\text{run}))$

For any state if the transaction enters the undo state then in some future state it will be ended or redone

18. $\mathbf{AG}((\mathit{Transaction}.\text{state}{=}\text{undo}) \rightarrow \mathbf{EF}(\mathit{Transaction}.\text{state}{=}\text{ended} \vee \mathit{Transaction}.\text{state}{=}\text{started}))$

For any state if the transaction is rolled-back then in some future state the operations already done will be undone or redone

19. $\mathbf{AG}((\mathit{Transaction}.\text{state}{=}\text{roll\_back}) \rightarrow \mathbf{EF}(\mathit{Transaction}.\text{state}{=}\text{undo} \vee \mathit{Transaction}.\text{state}{=}\text{redo}))$

For any state if the sign is valid and there is money available in some future state the bank transfer will be executed

20. $\mathbf{AG}((\mathit{CheckSign}.\text{state}{=}\text{success} \wedge \mathit{Balance}{=}\text{available}) \rightarrow \mathbf{EF}(\mathit{BankTransfer}{=}\text{success}))$

Eventually if the bank transfer starts then it will end in any case

21. $\mathbf{AF}((\mathit{BankTransfer}{=}\text{run}) \rightarrow \mathbf{AG}(\mathit{BankTransfer}{=}\text{success} \vee \mathit{BankTransfer}{=}\text{error}))$

For any state if the information is critical then it is possible to add the Signature

22. $\mathbf{AG}((\mathit{TypeInfo}{=}\text{critical}) \rightarrow \mathbf{EF}(\mathit{AddSign}{=}1))$

In some future state there might be not critical information

23. $\mathbf{EF}(\mathit{TypeInfo}{=}\text{not\_critical})$

For any state, if the transaction is successful then the system will update its resource in some future state

24. $\mathbf{AG}((\mathit{Transaction}.\text{state}{=}\text{commit}) \rightarrow \mathbf{EF}(\mathit{Transaction}.\text{state}{=}\text{update}))$

For any state if the bank transfer is successful then transaction will end in commit in some future state

25. $\mathbf{AG}((\mathit{BankTransfer}{=}\text{success} \wedge \mathit{CheckSign}.\text{state}{=}\text{success} \wedge \mathit{GetReq}.\text{state}{=}\text{success}) \rightarrow \mathbf{EF}(\mathit{Transaction}.\text{state}{=}\text{commit}))$

Whatever happens, the bank transfer will not be executed if there is no credit available

26. $\mathbf{AG}((\mathit{Balance}{=}\text{not\_available}) \rightarrow \mathbf{AG}\neg(\mathit{BankTransfer}{=}\text{success}))$

Error Management

The following formulas are necessary to prevent errors in the system behavior. The properties require that the specified situations should not occur, in fact they are formulated as the negation of the undesirable event.

There does not exist a future state in which an error occurs and the transaction is successful

27. $\neg$ **EF**((*GetReq*.state=error) $\wedge$ (*Transaction*.state=commit))

There does not exist a future state in which an error occurs during the transaction and it is successful

28. $\neg$ **EF**((*Transaction*.state=error) $\wedge$ (*Transaction*.state=commit))

The following two formulas ensure that, if the sign is not authentic, then the transaction will not end in commit

There does not exist a future state in which an error occurs in the sign verification and the transaction ends successfully

29. $\neg$ **EF**((*CheckSign*.state=error) $\wedge$ (*Transaction*.state=commit))

There does not exist a future state in which the information about the sign state that it is not valid and the transaction ends successfully

30. $\neg$ **EF**(*InfoSign*=not_valid $\wedge$ *Transaction*.state=commit)

The following two formulas ensure that if an error occurs during the bank transfer then the transaction will not end in commit

There does not exist a future state in which an error occurs in the bank transfer and the transaction ends in commit

31. $\neg$ **EF**((*BankTransfer*=error) $\wedge$ (*Transaction*.state=commit))

There does not exist a future state in which there is not credit available and transaction arrives in the commit state

32. $\neg$ **EF**(*Balance*=not_available $\wedge$ *Transaction*.state=commit)

6. Refinement and Evaluation of the Model

SMV was used to test the above specifications with respect to the system model that was synthesized, which is shown in Fig. 2 as a graph model.

This section discusses the size and variation in the model, the computational resources required to check those variations, and the counterexamples that were found during the verification process.

In the first-stage test, we modeled the transaction process as well behaved. As we might expect, we found no counterexamples in this predictable situation. A base-ease model was established and then models containing more details were tested. The level of detail was increased incrementally to search for faults that simpler models might not reveal and to observe the effects of model complexity on computational resources. Model complexity was increased by either adding more processes to the model or by modeling additional behaviors.

The aim was to verify that the system performs the transaction correctly or that the previous state is recovered when the transaction is unsuccessful.

To test the model validity we simulated some abnormal behavior during a transaction. In a business transaction, a critical issue is the signature authentication. We found counterexamples when we introduced somewhat abnormal behavior in this phase not unusual for b-2-b transactions.

We modeled a process *Checksign* for the signature authentication to perform the following steps:

- send a request to the certification authority and wait for an acknowledge message
- decrypt the document using the same algorithm of the sender.

The following are CTL formulas describing the undesired events that could happen if an error occurs during the two phases:

1. $\neg$ **EF**((*CheckSign*.state=error) $\wedge$ (*Transaction*.state=commit))
2. $\neg$ **EF**(*InfoSign*=not_valid $\wedge$ *Transaction*.state=commit)

Formulas 1 and 2 specify that the following conditions are never true simultaneously: 1) the sig-

nature is not valid and the *Transaction* reaches the commit state; 2) *Checksign* process is in error and the *Transaction* reaches the commit state.

The model checker returned the three counter examples. This was due to the logical model of the transaction process. In the adopted model the composition of the processes is interleaved: a step represents a step by exactly one component. For this reason, when SMV runs processes, it does not consider their concurrent execution and this might determine some errors.

To avoid the previously described undesired events it was necessary to introduce a further test on the state of each process. Once accomplished that all the conditions are verified, the process will start. This refinement of the model did not reveal incorrect behavior.

We simulated another error condition in the process performing a bank transfer. The following property verifies that the bank transfer will not be executed if the signature is false:

3. $\neg$ **EF**(*InfoSign*=not_valid $\wedge$ (*BankTransfer*=run))

Once more we use the negation of the error condition to ensure that the undesired event will not ensue.

The following 2 specifications state that if the desirable properties of a transaction are not satisfied, the transaction cannot end with a commit state. ACID properties of a transaction state that resources should be in a consistent and durable state after the execution of the transaction. This means that the transaction cannot be in the commit state if the transfer was not successful.

4. $\neg$ **EF**((*BankTransfer*=error) $\wedge$ (*Transaction*.state=commit))
5. $\neg$ **EF**(*Balance*=not_available $\wedge$ *Transaction*.state=commit)
6. $\neg$ **EF**((*Transaction*.state=error) $\wedge$ (*Transaction*.state=commit))

The last specification ensures that the transaction will never be in the commit state if an error occurs.

7. Conclusion

We used model checking for modeling the behavior of applications for business transactions. The model created was checked against temporal logic specifications, which identify the desired system behavior. Through symbolic model checking, counterexamples for some of these specifications were found. Many of these counterexamples would have been difficult to identify through conventional fault identification methods, because they were a result of multiple events occurring concurrently or sequentially. In many cases, a counterexample may point out a flaw in the model, which must be corrected. For this reason, the counterexamples helped in refining the model by adding further details; the final model ensures a reliable level of trust, since the model checker tests all patterns which are implicitly defined in the model.

Acknowledgements

This work has been carried out within the POP-EV funded project SFIDA3 (*Servizi di Firma Digitale Applicati ad Aziende e Amministrazioni*).

References

[1] R. E. BRYANT, (1986), Graph-based algorithms for Boolean function manipulation, *IEEE Trans. on Computers*, 35(8), 677–691.

[2] R. E. BRYANT, (1992) Symbolic Boolean manipulation with Ordered Binary-Decision Diagrams, *ACM Computing Surveys*, 24(3), 293–318.

[3] E. M. CLARKE, E. A. EMERSON, (1981) Design and synthesis of synchronization skeletons using branching time temporal logic.

[4] E. M. CLARKE E. A. EMERSON A. P. SISTLA, (1986) Automatic Verification of Finite-State Concurrent System using Temporal Logic Specifications. In *ACM Transaction on Programming Languages and systems*, vol. 8, N. 2.

[5] E. M. CLARKE, O. GRUMBERG AND D. A. PELED, (1999) *Model Checking*, The MIT Press.

[6] E. M. CLARKE AND J. M. WING, (1996) Formal Methods: state of the art and future directions, *ACM Computing Surveys*, 28(4), 1–22.

[7] J. CONALLEN, (1999) Modeling Web Application Architectures with UML, *Comm. of the ACM*, vol. 42, no. 10, 63–70.

[8] E. A. EMERSON, (1996) Automated temporal reasoning about reactive systems. In G. Goods and J. Hartmanis and J. van Leeuwen, eds., *Logic for concurrency: structure versus automata*, 41–92. Moller Birtwistle.

[9] M. R. A. HUTH, M. D. RYAN, (1999) *Logic in Computer Science. Modeling and reasoning about sytems*, Cambridge University Press.

[10] D. JUTLA, P. BODORIK, C. HAJNAL, C. DAVIS, (1999) Making Business Sense of Electronic Commerce, *IEEE Computer*, 67–75.

[11] J. P. KATOEN, (1999) *Concepts, Algorithms and Tools for Model Checking*, Friederik Alexander Universitat Erlangen – Nürnberg.

[12] S. KORPER, J. ELLIS, (2000) The E-commerce Book, Academic Press.

[13] L. LAMPORT, (1975) Proving the correctness of multiprocess programs. *IEEE Trans. Soft Engin.*, vol. 3, 125–143.

[14] D. W. MANCHALA, (2000) E-Commerce Trust Metrics and Models, *IEEE Internet Computing*, vol. 3–4, 36–44.

[15] Z. MANNA AND A. PNUELI, (1981) Verification of concurrent programs, Part1: the temporal framework, *Technical report, Stanford University Department of Computer Science*.

[16] F. MANOLA, (1999) Technologies for a Web Object Model, *IEEE Internet Computing*, vol. 1–2, 38–47.

[17] K. L. MCMILLAN, (1992) Symbolic Model Checking — an approach to state explosion problem, *Ph.D. thesis*, SCS, Carnegie-Mellon University.

[18] A. PNUELI, (1977) The Temporal Logic of Programs. In *Procedings of* 18^{th} *IEEE Symposium on the Foundations of Computer Science*, 46–57.

[19] SMV system draft.Available at: `http://www.cs.cmu.edu/ modelc|heck`.

Received: June, 2001
Accepted: September, 2001

Contact address:
Eugenio Di Sciascio, Ph.D.
Associate Professor
Dip. Elettrotecnica ed Elettronica, Politecnico di Bari
Via E. Orabona 4, I-70125 Bari, Italy
Phone: +390805963641
Fax: +390805963410
e-mail: disciascio@poliba.it, disciascio@acm.org
Web: http://www-ictserv.poliba.it/disciascio

EUGENIO DI SCIASCIO received the laurea "cum laude" degree in electronic engineering from University of Bari in 1989 and the Ph.D. degree in computer science in 1994 from Technical University of Bari. In 1992 he joined the University of Lecce as an assistant professor. He is currently an associate professor of Information Systems at Technical University of Bari. His research interests include image and video processing, multimedia information retrieval, knowledge-based systems for E-commerce. In these areas he has published several papers in international journals and conferences.

FRANCESCO M. DONINI got the Master's degree in electronics enginnering from the University of Rome "La Sapienza" in 1988. He got the Ph.D. in computer science from the same university in 1992. From 1991 to 1998 he was researcher/assistant professor at the Department of Computer and System Science of University of Rome "La Sapienza". Since 1998 he is associate professor at Technical University of Bari. He is coauthor of many papers in international journals, as well as international conferences. The paper "Tractable concept languages", presented at the conference IJCAI-91 (1991), received the best paper award. He is responsible for local university research projects since 1996, and for CNR research projects, too. He is editor of the area "Concept–Based Knowledge Representation" in the journal ETAI — Electronic Transactions on Artificial Intelligence — published by the Royal Swedish Academy.

MARINA MONGIELLO received the laurea "cum laude" degree in computer science from the University of Bari in 1993 and the Ph.D. degree in electronics engineering from the University of Catania in 2001. She is currently a research assistant at Technical University of Bari. Her research interests include multimedia information retrieval, knowledge-based systems for E-commerce and model checking of systems for E-commerce.

GIACOMO PISCITELLI was born in Bari, Italy, in January 1943. In 1966 he received the degree in physics with honors from the University of Bari. Currently, he is Professor of operating systems at the Department of Electrical and Electronic Engineering, Technical University of Bari. His primary areas of research and teaching are operating systems, information systems, software engineering.