ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/267257976

Formal-CAFE Methodology: an E-commerce System's Case Study

Article · October 2002

CITATIONS
0

READS
387

5 authors, including:

Adriano C. M. Pereira
Federal University of Minas Gerais
161 PUBLICATIONS **1,983** CITATIONS
SEE PROFILE

Mark A. J. Song
Pontifícia Universidade Católica de Minas Gerais
66 PUBLICATIONS **166** CITATIONS
SEE PROFILE

Wagner Meira Jr.
Federal University of Minas Gerais
584 PUBLICATIONS **10,249** CITATIONS
SEE PROFILE

Sergio Campos
Federal University of Minas Gerais
123 PUBLICATIONS **2,911** CITATIONS
SEE PROFILE

All content following this page was uploaded by Adriano C. M. Pereira on 08 December 2014.

The user has requested enhancement of the downloaded file.

Formal-CAFE Methodology: an E-commerce System's Case Study

Adriano Pereira, Mark Song, Gustavo Gorgulho, Wagner Meira Jr., and Sérgio Campos

Universidade Federal de Minas Gerais
Department of Computer Science
Belo Horizonte, Minas Gerais, Brazil
{adrianoc, song, gorgulho, meira, scampos}@dcc.ufmg.br

Abstract

Electronic commerce is an important application that has evolved significantly recently. However, electronic commerce systems are complex and difficult to be correctly designed. Currently, most approaches are *ad-hoc*, and frequently lead to expensive, unreliable systems that may take a long time to implement due to the great amount of errors. Moreover, guaranteeing the correctness of an e-commerce system is not an easy task due to the great amount of scenarios where errors occur, many of them very subtle. Such task is quite hard and laborious if only tests and simulation, common techniques of system validation, are used. In this work we presents a methodology that uses formal-method techniques, specifically *symbolic model checking*, to design electronic commerce applications and to automatically verify that these designs satisfy properties such as atomicity, isolation, and consistency. Using the proposed methodology, the designer is able to identify errors early in the design process and correct them before they propagate to later stages. Thus, its possible to generate more reliable applications, developed faster and at low costs. In order to demonstrate the applicability and feasibility of the technique, we have modeled and verified an English auction web site in which multiple buyers compete for product items. The proposed method can be applied in general e-commerce systems, where the business rules can be modeled by state transitions of the items on sale. We are currently studying other features of electronic commerce systems that we have not yet formalized, as well as the possibility of generating the actual code that will implement the system from its specification. In this context, we have been developing a set of design patterns to be used in the design and verification process of e-commerce systems. The idea is to define a model checking pattern hierarchy, which specifies patterns to construct and verify the formal model of e-commerce systems. We consider this research the first step to the development of a framework, which will integrate the methodology, an e-commerce specification language based on business rules, and a model checker. A future research is to apply our methodology in other application areas, such as mobile e-commerce and telecommunications.

Keywords Electronic Commerce, Formal Methods, Model Checking, Software Engineering, Software Verifi cation, Design Methodologies.

1 INTRODUCTION

One of the most promising uses of the Web is for supporting commercial processes and transactions. One of the advantages of the use of the Web in electronic commerce is that it allows one-to-one interaction between customers and vendors through automated and personalized services. Furthermore, it is usually a better commercialization channel than traditional ones because its costs are lower and it can reach an enormous potential customer population.

E-commerce has become a popular application. It makes the access to goods and services easy and has revolutionized the economy as a whole. In general, we can defi ne electronic commerce as the use of the network resources and information technology to ease the execution of central processes performed by an organization.

As new e-commerce services are created, new types of errors appear, some unacceptable. We define error as any unexpected behavior that occurs in a computer program. A typical error that may occur in a site is to allow two users to buy the same item. Nowadays, there is a consensus that the occurrence of errors in sites is a major barrier to the growth of e-commerce, as it may bring damages for the users and for the site, depending on its nature.

However, guaranteeing the correctness of an e-commerce system is not an easy task due to the great amount of scenarios where errors occur, many of them very subtle. Such task is quite hard and laborious if only tests and simulation, common techniques of system validation, are used.

Formal methods consist basically of the use of mathematical techniques to help in the documentation, specification, design, analysis, and certification of computational systems. The use of formal methods, in special model checking, is sufficiently interesting and promising once it consists of a robust and efficient technique to verify the correctness of several system properties, mainly regard to identification of faults in advance.

Formal methods are used in Computer Science mainly to improve the quality of the software and hardware or to guarantee the integrity of critical systems. In general, formal methods are used forsake of modeling, formal specification and formal verification of the system, that are the basic processes of the model checking. The design and implementation are modeled considering the features that will be handled in the formal specification, making possible to demonstrate its conformity through formal verification.

Formal methods embrace a variety of approaches that differ considerably in techniques, goals, claims, and philosophy. The different approaches to formal methods tend to be associated with different kinds of specification languages. Conversely, it is important to recognize that different specification languages are often intended for very different purposes and therefore cannot be compared directly to one another. Failure to appreciate this point is a source of much misunderstanding. In this work we use model checking, which is an interesting formal method technique to verify hardware and software systems using temporal logic.

Temporal logics have proved to be useful for specifying concurrent systems, because they can describe the ordering of events in time without introducing time explicitly. They were originally developed by philosophers for investigating the way that time is used in natural language arguments [18].

Model checking is a method for formally verifying finite-state concurrent systems. Specifications about the system are expressed as temporal logic formulas, and efficient symbolic algorithms are used to traverse the model defined by the system and check if the specification holds or not. Extremely large state-spaces can often be traversed in minutes.

Applying model checking to a design consists of several tasks, that can be classified in three main steps, as follows:

Modeling: consists of converting a design into a formalism accepted by a model checking tool .

Specification: before verification, it is necessary to state the properties that the design must satisfy. The specification is usually given in some logical formalism. For hardware and software systems, it is common to use *temporal logic*, which can assert how the behavior of the system evolves over time.

An important issue in specification is *completeness*. Model Checking provides means for checking that a model of the design satisfies a given specification, but it is impossible to determine whether the given specification covers all the properties that the system should satisfy. This problem illustrates how important a methodology is to conceive a better specification in terms of *completeness*.

Verification: ideally the verification is completely automatic. However, in practice it often involves human assistance. One such manual activity is the analysis of the verification results. In case of a negative result, the user is often provided with an error trace. This can be used as a counterexample for the checked property and can help the designer in tracking down where the error occurred. In this case, analyzing the error trace may require a modification to the system and reapplication of the model checking algorithm.

This paper presents a new methodology to design e-commerce systems applying model checking. The Section 2 defines some important concepts about model checking and its application in e-commerce systems. In the Section 3, our proposed methodology is explained, and Section 4 shows an example of its use, an English auction case study. The Section 5 analyzes the related works, and Section 6 presents some conclusions and future work.

2 MODEL CHECKING OF E-COMMERCE SYSTEMS

Most electronic commerce systems can be modeled using a few entities: the products being commercialized such as books or DVDs, the agents that act upon these products such as consumer or seller, and the actions that modify the state of the product such as reserving or selling an item.

Similarly to traditional commercial systems, the main entity of electronic commerce is the product that is transactioned. For each product being commercialized there are one or more items, which are instances of the product. Each item is characterized by its life cycle, which can be represented by a state-transition graph, i.e., the states assumed by the item while being commercialized and the valid transitions between states. Examples of states are *Reserved* or *Sold*. The item's domain is the set of all states the item can be.

The entities that interact with the e-commerce system are called agents. Examples of agents are buyers, sellers and the store manager. The agents perform actions that may change the state of an item, that is, actions correspond to transitions in the life cycle graph. Put an item in the basket or cancel an item's reserve are examples of actions.

Services are sequences of actions on products. While each action is associated with an item and usually comprises simple operations such as allocating an item for future purchase, services handle each product as a whole, performing full transactions. Purchasing a book is an example of a service, which consists of paying for the book, dispatching it, and updating the inventory.

2.1 Business Rules

An e-commerce system can be described by its business rules. A business rule is a norm, denoted property, which specifies the functioning of an e-commerce application. For example, a rule can describe that an item can only be reserved if it is available. To specify this property, a developer would have to translate this informal requirement into the following CTL formula: AG (((state = available) & (action = reserve) & (inventory > 0)) $\rightarrow$ AX ((state = reserved) & (next(inventory) = inventory - 1))).

As we can see, the specification process will demand some expertise in formal methods. We contend that acquiring this level of expertise represents a substantial obstacle to the adoption of the methodology. We propose to overcome this by using an *specification pattern system*.

Although formal specification and verification methods offer practitioners some significant advantages over the current state-of-the-practice, they have not been widely adopted. Partly this is due to a lack of definitive evidence in support of the cost-saving benefits of formal methods, but a number of more pragmatic barriers to adoption of formal methods have been identified [11], including the lack of such things as good tool support, appropriate expertise, good training materials, and process support for formal methods.

The recent availability of tool support for finite-state verification provides an opportunity to overcome some of these barriers [14]. Finite-state verification refers to a set of techniques for proving properties of finite-state models of computer systems. Properties are typically specified with temporal logics or regular expressions, while systems are specified as finite-state transition systems of some kind. Tool support is available for a variety of verification techniques including, for example, techniques based on model checking [21], bisimulation [26], language containment [17], flow analysis [15], and inequality necessary conditions [1]. In contrast to mechanical theorem proving, which often requires guidance by an expert, most finite-state verification techniques can be fully automated, relieving the user of the need to understand the inner workings of the verification process. Finite-state verification techniques are especially critical in the development of concurrent systems, where non-deterministic behavior makes testing especially problematic.

According to Dwyer [14], despite the automation, users of finite-state verification tools must still be able to specify the system requirements in the specification language of the tool. This is more challenging than it might appear at first.

Acquiring this level of expertise represents a substantial obstacle to the adoption of automated finite-state verification techniques and that providing an effective way for practitioners to draw on a large experience base can greatly reduce this

obstacle. Even with significant expertise, dealing with the complexity of such a specification can be daunting. In many software development phases, such as design and coding, complexity is addressed by the definition and use of abstractions. For complex specification problems, abstraction is just as important.

Dwyer [13, 14] developed a system of property specification patterns for finite-state verification tools based in the scope, order and occurrence of an event[1]. Each pattern has a scope, which is the extent of the program execution over which the pattern must hold. There are five basic kinds of scopes, as explained:

- Global: the entire program execution;
- Before: the execution up to a given event;
- After: the execution after a given event;
- Between: any part of the execution from one given event to another given event;
- After-until: like between but the designated part of the execution continues even if the second event does not occur.

The scope is defined by specifying a starting and an ending event for the pattern. For state-delimited scopes, the interval in which the property is evaluated is closed at the left and open at the right end. Thus, the scope consists of all states beginning with the starting state and up to but not including the ending state. A list of some patterns, with short descriptions, follows:

- Absence: a given event does not occur within a scope.
- Existence: a given event must occur within a scope.
- Bounded Existence: a given state must occur k times within a scope.
- Universality: a given event occurs throughout a scope.
- Precedence: an event P must always be preceded by an event Q within a scope.
- Response: a event P must always be followed by an event Q within a scope.
- Chain Precedence: a sequence of events $P_1, ..., P_n$ must always be preceded by a sequence of events $Q_1, ..., Q_n$.
- Chain Response: a sequence of events $P_1, ..., P_n$ must always be followed by a sequence of events $Q_1, ..., Q_n$.

Using this classification we could express the example explained in this section using the patterns of *universality*, *chain response* and the logical operators.

In the next subsections we present a brief background on model checking and CTL-formulas.

2.2 Symbolic Model Checking

Ensuring the correctness of the design at its earliest stage is a major challenge in any system development process. Current methods use techniques such as *simulation* and *testing* for design validation. Although effective in the early stages of debugging, their effectiveness drops quickly as the design becomes clear. A serious problem with these techniques is that they explore *some* of the possible behaviors of the system. We may never be sure whether the unexplored execution paths may contain fatal bugs. A very attractive alternative to simulation and testing is the use of *formal verification* approach, which explores exhaustively all possible behaviors of the system.

[1] In an e-commerce system, an event describes a set of actions

Symbolic model checking is a formal verification technique by which a desired behavioral property of a system can be verified over a model through exhaustive enumeration of all the states reachable by the application and the behaviors that traverse through them. The system being verified is represented as a *state-transition graph* (the model) and the *properties* (the behaviors) are described as formulas in temporal logic. Labels are associated to the values of the variables in the program, while transitions correspond to steps in the model.

A state is a snapshot of the system, capturing the values of the variables at a particular instant of time. An assignment of values to all variables defines a state in the graph. For example, if the model has three boolean variables a, b, and c, then (a=1,b=1,c=1), (a=0,b=0,c=1), and (a=1,b=0,c=0) are examples of possible states. The *symbolic representations* of these states are (a, b, c), $(\overline{a}, \overline{b}, c)$, and $(a, \overline{b}, \overline{c})$, respectively, where a means that the variable is true in the state and $\overline{a}$ means that the variable is false. Boolean formula over variables of the model can be true or false in a given state. Note that the value of a boolean formula in a state is obtained by substituting the values of the variables into the formula for that state. For example, the formula $a \vee c$ is true in all the three states discussed above.

The graph representation can be a direct consequence of this observation. We can use a boolean formula to denote the set of states in which that formula is satisfied. For example, the formula *true* represents the set of all states, the formula *false* represents the empty set with no states, and the formula $a \vee c$ represents the set of states in which a or c are true. Notice that individual states can be represented by a formula with exactly one proposition for each variable in the system. For instance, the state $s = (a, \overline{b}, c)$ is represented by the formula $a \wedge \neg b \wedge c$. We say that $a \wedge \neg b \wedge c$ is the formula associated with the state s. Because symbols are used to represent states, algorithms that use this method are called symbolic algorithms.

$$(\neg a, \neg b, \neg c) \longrightarrow (\neg a, b, \neg c) \quad \Rightarrow \quad \neg a \wedge \neg b \wedge \neg c \wedge \neg a' \wedge b' \wedge \neg c'$$

Figure 1: Example of a transition and its symbolic representation.

Transitions can also be represented by boolean formulas. A transition $s \rightarrow t$ is represented by using two distinct sets of variables, one set for the current state s and another set for the next state t. Each variable in the set of variables for the next state corresponds to exactly one variable in the set of variables for the current state. For instance, if the variables for the current state are a, b, and c, then the variables for the next state are labeled a', b', and c'. Let f_s be the formula associated with the state s and f_t with the state t. Then, the transition $s \rightarrow t$ is represented by $f_s \wedge f_t$. The meaning of this formula is the following: there exists a transition from state s to state t if and only if the substitution of the variable values for s in the current state and those of t in the next state yields *true*. For example, a transition (Figure 1) from the state $(\overline{a}, \overline{b}, \overline{c})$ to the state $(\overline{a}, b, \overline{c})$ is represented by the formula $\neg a \wedge \neg b \wedge \neg c \wedge \neg a' \wedge b' \wedge \neg c'$.

2.3 The Computation Tree Logic - CTL

Computation tree logic is the logic used to express properties that will be verified by the model checker. *Computation trees* are derived from state transition graphs. The graph structure is unwound into an infinite tree rooted at the initial state, as seen in figure 2. Paths in this tree represent all possible computations of the program being modeled.

Formulas in CTL refer to the computation tree derived from the model. It is classified as a *branching time* logic, because it has operators that describe the branching structure of this tree. Formulas in CTL are built from atomic propositions, boolean connectives $\neg$ and $\wedge$, and *temporal operators*. Each of these operators consists of two parts: a path quantifier followed by a temporal quantifier. Path quantifiers describe the branching structure - they indicate that the property should be true in *all* paths from a given state (**A**), or *some* path from a given state (**E**). The temporal quantifier describes how events should be ordered with respect to time for a path specified by the path quantifier. Examples of temporal quantifiers and their informal meanings are: **F** φ, meaning that φ holds sometime in the future; **G** φ, meaning that φ holds globally on the path; **X** φ, meaning that φ holds in the next state. Some examples of CTL formulas are given below to illustrate the expressiveness of the logic.

- $\mathbf{AG}(req \rightarrow \mathbf{AF}\ ack)$: It is always the case that if the signal req is high, then eventually ack will also be high.

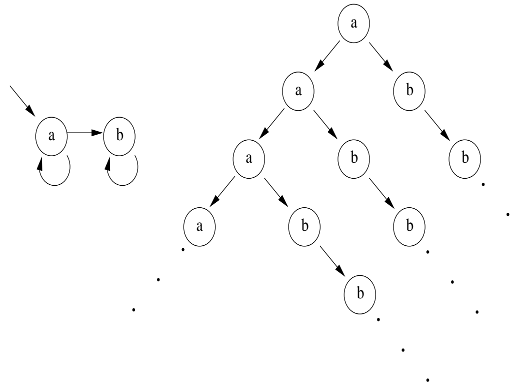

Figure 2: State transition graph and corresponding computation tree.

- $\mathbf{EF}(started \wedge \neg ready)$: It is possible to get to a state where $started$ holds but $ready$ does not hold.

In the next section we explain our methodology to incrementally design verifiable e-commerce systems.

3 THE FORMAL-CAFE METHODOLOGY

There are many types of e-commerce applications, such as digital library, virtual bookstore, auction sites and others. The difference between them are their nature and their business rules. Some business rules are common, for example: an item should not be sold to more than one customer. On the other hand, there are many other rules specific to the application, as to allow or not the reservation of an item, to provide supply control, or to define priority to transactions executed concurrently.

The proposed methodology, an extension of the *CAFE* methodology [19], consist of a way to design e-commerce systems to apply model checking. *CAFE* methodology explains how to specify an e-commerce system and we consider the user should know some formal language, such as SMV [20], to build its model.

Our methodology is incremental and divided into four major levels. Its relevant to emphasize that these organization were adopted in order to simplify the design specification, but the designer may employ another organization.

The first level, defined as conceptual, embodies the business rules and the definition of the e-commerce system to be designed. As many details the designer specifies, as easier would be to apply the methodology and achieve good results in the verification process.

The second level, called application, models the life cycle of the item that is commercialized, identifying the types of operations (called actions, as we refer to henceforth) that are performed on it and change its state.

The third one, named functional, models the services provided by the system and the concept of multiple items are introduced.

The last level contemplates the components of the system and the user's interaction with them. It completes the scope of the system, modeling its architecture, so we called it the architectural level.

3.1 Properties of an e-commerce system

Properties may be described, for examples, as formulas in CTL, Subsection 2.3. CTL-formulas are built from atomic propositions, boolean connectives and temporal operators.

For example, a rule can describe that an item can only be reserved if it is available. To specify this property, a developer would have to translate this informal requirement into the following CTL-formula: AG (((item_state = Available) & (service = Reserve) & (product_inventory $>$ 0) & (next(product_inventory) = product_inventory - 1)) $\rightarrow$ AX ((item_state = Reserved))).

As we can see the specification process demands some expertise in formal methods. We contend that acquiring this level of expertise represents a substantial obstacle to the adoption of the methodology. So, to overcome this problem, we suggest to use a *specification pattern system*.

Thinking about an e-commerce application model, we realize that the first important property to verify is completeness. This property guarantees that the model is consistent, by asserting that all states and actions are achieved.

To express this property we can use the property pattern of *Existence*. Additionally it is necessary to define the scope as *after*, considering that "Exists in the Future" means "after current state/event".

Transitivity is a property which defines the next state to be achieved after the occurrence of an event in the current state. It is necessary to check its veracity to guarantee the correct execution of the services that satisfy the business rules.

Most properties to be verified in e-commerce systems relate to transactions. A transaction is an abstraction of an atomic and reliable sequence of operations executed. Transaction processing is important for almost all modern computing environments that support concurrent processing. In an electronic commerce server, a transaction consists of a sequence of actions affecting the existing items, each action potentially modifying the state of the item. One of the most important properties that must be satisfied in this context is guaranteeing that the transactions being executed are consistent, that is, showing that the concurrency control mechanism implemented is correct and that concurrent transactions do not interfere with each other. In other words, we must check that transactions are atomic.

We have verified three types of properties that relate to the consistency of transactions: *Atomicity* - a transaction must be finished or not started, that is, if it does not finish, its effects have to be undone; *Consistency* - the state of the product must remain coherent at all times; *Isolation* - the result of one transaction must not affect the result of another concurrent transaction.

In the next subsections we detail the levels of the formal methodology, using real examples of e-commerce business rules to explain how this properties can be checked.

3.2 Conceptual Level

Formally, we characterize an e-commerce system by a tuple $< P, I, D, Ag, Ac, S >$, where P is the set of products, I is the set of items, D is the set of product domains, Ag is the set of agents, Ac is the set of actions and S is the set of services.

Products are sets of items, that is, $i \in I$ means that $i \in p, p \in P$. The products partition the set of items, that is, every item belongs specifically to a single product. Formally, $I = \bigcup_{\forall p \in P} p$ and $p_i \cap p_j = \emptyset$ for $i \neq j$. Domains are associated with items, that is, each item i is characterized by a domain D_i. Two items of the same product have the same domain, i.e., for all items $i, j \in I$, there is a product p such that if $i \in p$ and $j \in p$, then $D_i = D_j$.

Each action is associated with a transition in the state-transition graph of the item and is defined by a tuple $< a, i, tr > \in Ac$, where $a \in Ag$ is the agent that performs the action, and $i \in I$ is the item over which the action is performed, and $tr \in D_i X D_i$ is the transition associated with the action. In our model, the actions performed on a given item are totally ordered, that is, for each pair of actions x and y, where i_x and i_y are the same, either x has happened before y or y has happened before x. Services are defined by tuples $< p, A >$, where $p \in P$ and $A = a_1, a_2, \ldots$ is a sequence of actions such that if $a_i = (d_1, d_2)$, $a_{i+1} = (d_3, d_4)$ then $d_2 = d_3$ $\forall i, d_i \in D_j$ where D_j is the domain of an item from p.

The items are modeled by their *life cycle graphs*, which represent the state each item can be in during its life cycle in the system. An example of a life cycle graph can be seen in figure 3. States in this graph are possible states for the item such as *Available*, or *Reserved*. Transitions represent the effect of actions such as reserving an item or buying it.

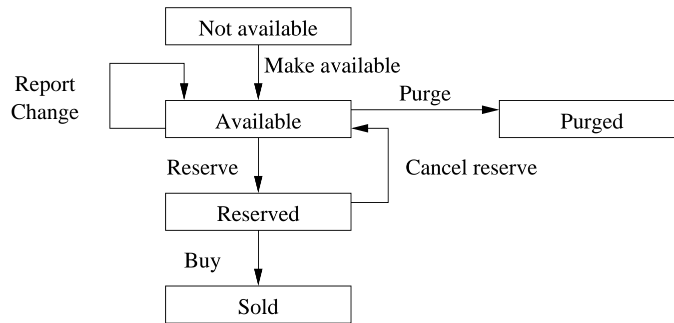

Figure 3: The life cycle graph of product's item

Each item from I has several attributes, including the associated product, its state, and other characteristics. Finally, the agents are represented by concurrent processes that execute services, which are sequences of transitions on the state-transition graphs.

In this model, each global state represents one state in each product life cycle graph, and transitions model the effects of actions in the system. Therefore, paths in the global graph represent events that can occur in the system. The life cycle of the product is the set of all life cycles of its items.

In this first level we do not have properties to verify, but we are interested in prepare the designer to specify the e-commerce application as best as possible. This decision will ease the work to perform in the next levels of the methodology.

3.3 Application Level

This level describes the e-commerce system in terms of the life cycle of the items. It is necessary to identify the states of an item, its attributes, the set of actions that could be executed on it and the effects caused by them and the agents that execute these actions. Here we are not interested in the functionalities of the web site and the architecture of them yet.

In this level it is important to verify the completeness property of the e-commerce model. Here, it is important to observe that there are only actions and states. Actions, by definition, are transactional, so the atomicity, consistency and isolation are guaranteed. Transitivity is related to the functionalities, so it will be important only in the next level, where there are services being executed in the model.

To check the completeness property of the business model, we use the CTL-formulas below, where S consists of all the states presented in the application model and A, the universe of actions.

```
EF (state = <S>)           EF (action = <A>)
```

The Figure 4 illustrates the first level of our methodology. As this figure shows there are agents (Seller and Buyer) that represent the consumer and the supplier of the system. There is an item, which has a set of states. The agents execute actions that could affect the item's state.

3.4 Functional Level

This level introduces the product, composed by zero (the product is not available) or more items. The designer determines the operations the agents can perform, denoted as services. A service is executed on products and its effects might change or not the state of it and its items. The focus of this level is to define clearly how the services are executed and what happened with the product and its items in this case.

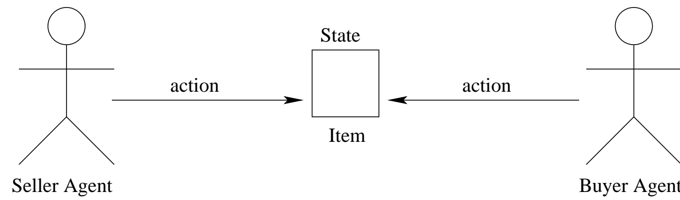

Figure 4: The First Level of the Methodology

In this level it is important to verify the transitivity property of the model. The agents execute services that change the state of the item. This state must be consistent with the life cycle of the item and the related business rule associated with it.

Some examples of transitivity are:

```
AG ((item_state = Not_Available & service = Make_Available) ->
   AX (item_state = Available)
or
AG ((item_state = Available & service = Purge) -> AX (item_state = Purged)
```

In this level it is important to verify the atomicity, consistency and isolation properties either. It is essential to check the consistency between the state of the product and its items in a given moment. Moreover, there are agents performing services concurrently, which may cause the system to achieve an invalid state. Therefore the isolation property must be guaranteed.

To become clear, we give some examples of this properties[2]. First we give an example of atomicity. if an item is available and a *Reserve* service is performed by a buyer agent and granted by the server, the item must be reserved in the next state and the inventory must be decremented.

```
AG ((item_state = Available & service = Reserve & product_inventory = 1) ->
AX (item_state = Reserved & product_inventory = 0))
```

Examples of consistency properties can be seen below:

- If the inventory is zero, then no item should be available.

  ```
  AG (product_inventory = 0 -> !product_state = Available)
  ```

- Conversely, if there is inventory, at least one item must be available.

  ```
  AG (product_inventory > 0 -> product_state = Available)     or
  AG (product_inventory > 0 -> (item1_state = Available) |
     (item2_state = Available))
  ```

An example of isolation property could be: if there are two items available and two buyers reserve these items simultaneously, the inventory must be zero in the next step.

```
AG ((buyer1_service = Reserve & buyer2_service = Reserve &
     product_inventory = 2) -> AX (product_inventory = 0))
```

This level is depicted in Figure 5. As it shows, there are agents that execute services, that could modify the product's state. Some of these services, as a *Reserve* of an instance of the product, change the state of the item either.

It is important to notice that the properties validated in the second level should retain their validity in the third one and so on. The verification of the properties should be incremental as well as the methodology proposed.

[2]The actual properties verified are slightly different than the ones presented here, which have been simplified for readability.

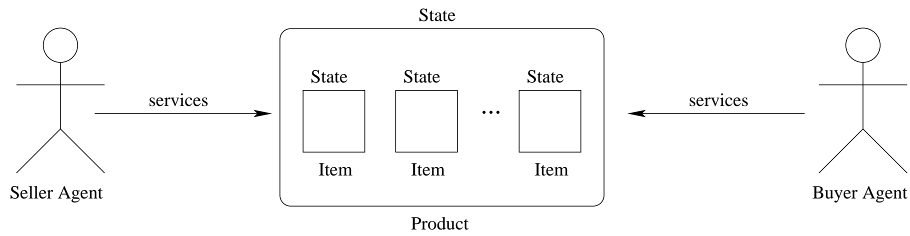

Figure 5: The Second Level of the Methodology

3.5 Execution or Architectural Level

This level specifies the system in terms of its components and the way they interact with each other. It is important to emphasize that this level encompasses the other ones, completing the specification of the system and describing its architecture.

In this stage, the model is more complex, contemplating the components of the system's architecture. We identify the atomicity properties that should be verified and other consistency and isolation rules. The transactional properties verified in the other levels should be checked again, as the transitivity properties.

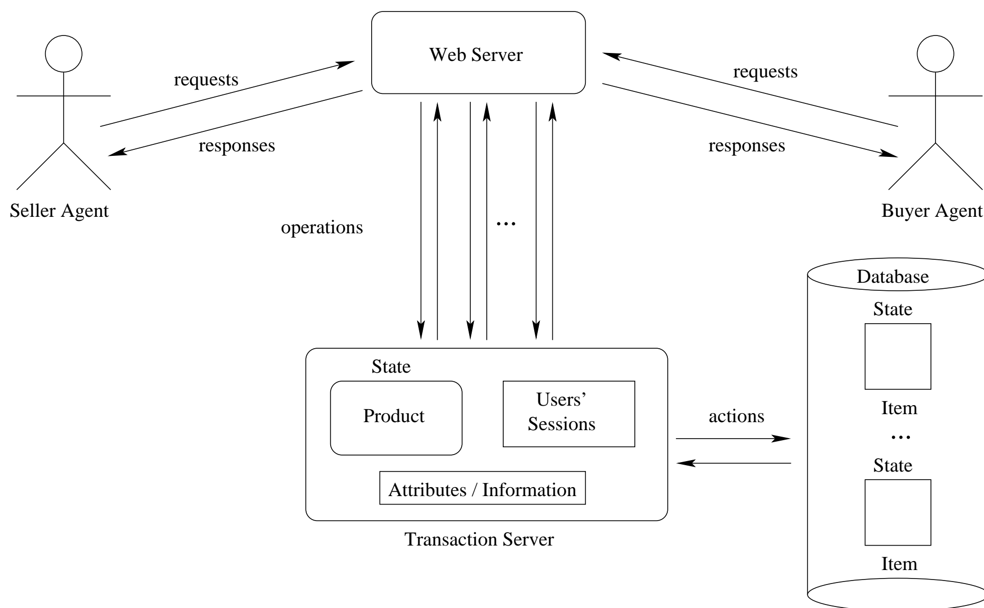

Figure 6: The Third Level of the Methodology

As showed by the Figure 6, we introduce the components of the e-commerce system: the web server, the transaction server and the database. There are agents that submit requests to the web server, which translate them into operations to the transaction server. These operations, named services, are executed by this server, sometimes performing action on the items. This level is important because it enables the designer to get a specification closer to the real implementation he wants to develop.

We validate this methodology trough a case study, which models an English auction web site. In [25] we had presented another case study, a virtual store. As a result of that case study, we were able to detect a serious error, that violated the isolation property, causing the same item to be sold twice. It occurred because two buyer agents tried to acquire the product at the same time and there was only one item available. The proposed methodology applied to this case study enable us to

identify this error and fix it to guarantee the correctness of our design. In the next section we present the English auction case study.

4 CASE STUDY: AN ENGLISH AUCTION WEB SITE

In this section, another case study is presented, an English auction site, a popular web application. This is a common electronic business application in which most of the aspects that make such applications complex to design are present, such as multiple agents of different types that compete for access to products, products with more than one item and intermediate states for items (for example, one may reserve an item before buying it). We have used the NuSMV model checker [7, 8, 9] to perform this task.

William Vickrey [31] established the basic taxonomy of auctions based upon the order in which prices are quoted and the manner in which bids are tendered. He established four major (one sided) auction types.

The English Auction is the most common type of auction. The English format can be used for an auction containing either a single item or multiple items. In an English Forward auction, the price is raised successively until the auction closes. In an English Reverse auction the price is lowered until the auction closes. At auction close, the Bidder or Bidders declared to be the winner(s) are required to pay the originator the amounts of their respective winning bids. This case study considers the English Forward auction, also known as the open-outcry auction or the ascending-price auction. It is used commonly to sell art, wine and numerous other goods.

Paul Milgrom [22, 23, 24] defines the English auction in the following way. "Here the auctioneer begins with the lowest acceptable price (the reserve price: lowest acceptable price. Useful in discouraging buyer collusion.) and proceeds to solicit successively higher bids from the customers until no one will increase the bid. The item is "knocked down" (sold) to the highest bidder."

Contrary to popular belief, not all goods at an auction are actually knocked down. In some cases, when a reserve price is not met, the item is not sold.

Sometimes the auctioneer will maintain secrecy about the reserve price, and he must start the bidding without revealing the lowest acceptable price. One possible explanation for the secrecy is to thwart rings (subsets of bidders who have banded together and agree not to out-bid each other, thus effectively lowering the winning bid).

The next subsections present the English auction model created using the *Formal-CAFE* methodology.

4.1 Conceptual Level

An English Auction consists of an only seller and one or more buyers that want to acquire the item of the auction. The salesman creates this auction specifying:

- the init date of the auction.
- the finish date of the auction.
- minimum value (minimum value of the bid that is accepted).
- private value (optional attribute, that denotes the lesser value of the bid accepted by the salesman for concretion of the business).
- minimum increment (optional attribute, that denotes the minimum value between two consecutive bids).

The buyers might make bids as many as they want. The following rules are defined:

- the first bid's value must be equal or higher than the attribute minimum value.
- The bids must be increased at each iteration.
- Who wins the auction: the buyer who makes the higher bid until the end of the auction, and this bid must be equal or higher than the attribute private value, defined by the seller. If this attribute is not defined, the bid is the winner.

There are the following entities in the model:

- buyer agent;
- seller agent;
- transaction server and
- English auction server.

There still have the modules *web server* and *database*, but I abstracted them here, as they will be on the architectural level only.The next paragraphs present some high-level description of the entities.

MODULE English auction server: it is responsible to dispatch some events that controls the auction workflow. The important states that an auction should assume are:

- closed, to be initiated.
- opened without bids.
- opened with bids, but the private value has not been achieved.
- opened with bids and the private value has already been achieved.
- Finished without winner.
- Finished with winner.

Other attributes should be stored as the buyer id that wins an auction, number of bids made and their values, and so on.

MODULE buyer agent: represents the consumer, the person who wants to buy some product.The following actions could be executed by the buyer agent:

- get: show information about a specific auction.
- list: list the auctions.
- bid actions: actions related to bid as create, get and list.

In our model, the *Report* action represents two possibilities related to information about the auction: *get* and *list*. The bid actions are modeled as *Make Bid*.

MODULE seller agent: represents the seller, the person who wants to sell some product using the English auction mechanism. The following actions could be executed by the seller agent:

- get: show information about a specific auction.

- create: create a new auction.
- list: list the auctions.
- update: update the information of a specifi c auction.
- make available: a new item is added to the inventory.
- purge: an item is removed from the inventory.
- cancel auction: the current auction negotiation is canceled.

In our model, the *Report* action executed by the seller agent represent *get* and *list* actions described. The *create* functionality is represented by the action *Reserve in Auction*. In the same manner, *update* is described as *Change* action. The functionalities *make available*, *purge*, and *cancel auction* are represented by actions with its respective names.

MODULE transaction server: represents the server responsible for execute the actions of the agents and keep the users session state. It starts the English auction process, which could be represent by the init page (home) of the auction web site. When this state is achieved, the agents would execute any of the English auctions actions allowed, as previously described.

Considering the English auction rules, a bid would be accepted if:

- the auction is opened.
- the bid's value is greater than the minimum value.
- the bid's value is greater than the last one gave (considering the minimum increment, if it was defi ned by the seller).

In this case study, the life cycle graph of the product's item has the following states, as can be seen in fi gure 7: *Not Available*, *Available*, *Reserved in Auction*, *Sold in Auction* and *Purged*. The transitions in the graph can be seen in the fi gure. The global model of the English auction web site is a collection of life's cycle graphs and additional attributes represented by variables such as the inventory (the number of items available). Additional logic is needed to "glue" together the various life cycle graphs.

Finally, the agents are modeled as concurrent processes that perform actions. In the model there is one seller agent that represents the administrator of the store and one or more buyer agents that act as the customers. To illustrate how the methodology works in practice, we will present parts of the SMV code for the English auction web site.

As defi ned in the methodology, Section 3.2, conceptual level details the e-commerce system requirements.

The MODULE *English Auction Server* has the responsibility to control the auction process. To do it, there are some events it has to manage, defi ned in Table 1.

A set of business rules we identify in our study case are cited below:

- If the item is in the state *Not Available* and the action *Make Available* occurs, the next state is *Available*.
- If the item is in the state *Available* and the action *Purge* occurs, the next state is *Purged*.
- If the item is in the state *Available* and the action *Change* occurs, the next state is the same.
- If the item is in the state *Available* and the action *Report* occurs, the next state is the same.
- If the item is in the state *Available* and the action *Reserve in Auction* occurs, the next state is *Reserved*.
- If the item is in the state *Reserved in Auction* and the action *Cancel Auction* occurs, the next state is *Available*.

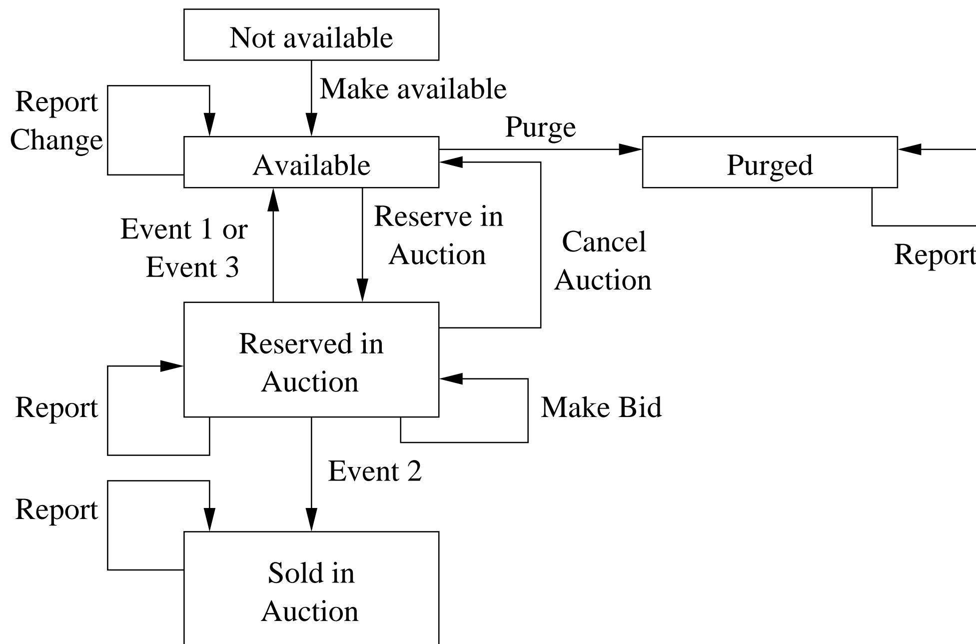

Figure 7: An English Auction Site - The life cycle graph of product's item

Id	Dispatch Condition	Result
1	fi nish date is achieved and the reserved value is not reached	the item in auction will be available
2	fi nish date is achieved and the reserved value is reached	the item will be sold to the owner of winner bid
3	fi nish date is achieved and nobody made bids	the item in auction will be available

Table 1: English Auction Events

- If the item is in the state *Reserved in Auction* and the action *Report* occurs, the next state is *Reserved in Auction*.
- If the item is in the state *Reserved in Auction* and the action *Event 1* occurs, the next state is *Available*.
- If the item is in the state *Reserved in Auction* and the action *Event 3* occurs, the next state is *Available*.
- If the item is in the state *Reserved in Auction* and the action *Event 2* occurs, the next state is *Sold in Auction*.
- If the item is in the state *Reserved in Auction* and the action *Make Bid* occurs, the next state is *Reserved in Auction*.
- If the item is in the state *Purged* and the action *Report* occurs, the next state is *Purged*.
- If the item is in the state *Sold in Auction* and the action *Report* occurs, the next state is *Sold in Auction*.
- The inventory of the product must be positive.
- If the inventory is positive, at least one item must be available.
- If the inventory is null, then the product must be not available.
- The actions *Reserve in Auction* and *Cancel Auction* must be atomic.
- If there are agents executing concurrently, their actions must be isolated.

- Events 1, 2, and 3 must be isolated.

It is important to emphasize that all business rules must be granted in the next levels to confirm the correctness of the case study.

4.2 Application Level

A module in SMV consists of a number of variables and their assignments. The main module consists of the parallel composition of each module. This is accomplished by instantiating each module in the main module shown as follow:

```
MODULE main

VAR
        it1: process item(1,buyer1.action, seller1.action, sys.event);
        buyer1: process buyer_agent(1,action);
        seller1: process seller_agent(1,action);
        sys: process system(event);
```

The process *system* is representing the module *English auction server*, which dispatches the events. As described in Section 3.3, the first important property to be verified is completeness. In this case study, we can do this through the specification written in CTL formulas:

```
EF (it1.state = Not Available)
EF (it1.state = Available)
EF (it1.state = Reserved in Auction)
EF (it1.state = Sold in Auction)
EF (it1.state = Purged)

EF (buyer1.action = Report)
EF (buyer1.action = Make Bid)
EF (buyer1.action = None)

EF (seller1.action = Make Available)
EF (seller1.action = Change)
EF (seller1.action = Purge)
EF (seller1.action = Reserve in Auction)
EF (seller1.action = Cancel Auction)
EF (seller1.action = None)

EF (sys.event = 1)
EF (sys.event = 2)
EF (sys.event = 3)
EF (sys.event = None)
```

These specifications should be consistent with the item's life cycle graph, as illustrated by Figure 7. In our model all of them were verified as *true*, certifying its completeness.

It is important to emphasize that we put the *None* action to represent the situation where the agents and system do not execute any action. This situation is frequently observed in web sites and this interval between two consecutive actions of an agent is known as "think time".

This version of the model has 4 modules (corresponding to 4 processes), which corresponds to 156 lines of SMV code, and 18 properties verifi ed.

Once we have checked this property, we continue the model, building the third level.

4.3 Functional Level

Continuing the process defi ned by the methodology we add new modules to the model, which represents the product and its items. Here, we are interested in verify some business rules related to services.

Initially, as described in Section 3.4, we have to check the transitivity properties of the model. We can perform this using the following CTL formulas:

```
AG (it1.state = Not Available & service = Make_Available) ->
    AX (it1.state = Available)

AG (it1.state = Available & service = Report) ->
    AX (it1.state = Available)
AG (it1.state = Available & service = Change) ->
    AX (it1.state = Available)
AG (it1.state = Available & service = Reserve in Auction) ->
    AX (it1.state = Reserved in Auction)
AG (it1.state = Available & service = Purge) -> AX (it1.state = Purged)

AG (it1.state = Reserved in Auction & service = Report) ->
    AX (it1.state = Reserve in Auction)
AG (it1.state = Reserved in Auction & service = Make Bid) ->
    AX (it1.state = Reserved in Auction)
AG (it1.state = Reserved in Auction & service = Event 1) ->
    AX (it1.state = Available)
AG (it1.state = Reserved in Auction & service = Event 2) ->
    AX (it1.state = Sold in Auction)
AG (it1.state = Reserved in Auction & service = Event 3) ->
    AX (it1.state = Available)
AG (it1.state = Reserved in Auction & service = Cancel Auction) ->
    AX (it1.state = Available)

AG (it1.state = Sold in Auction & service = Report) ->
    AX (it1.state = Sold in Auction)

AG (it1.state = Purged & service = Report) ->
    AX (it1.state = Purged)
```

To make easy to understand these representations, we abstracted of the items' id. Based on these transitivity properties and the business rules, its possible to include new propositions, which will restrict some transitions. This will make possible to verify the transactional properties of the model, such as atomicity, consistency and isolation.

Here, it is explained some transactional properties, beginning with atomicity. if an item is available and a reserve action is performed by a buyer, the item must be reserved in the next state and the state must be consistent with this or the service is not executed and the state is not modified.

```
AG ((state = Available & service = Reserve in Auction & inventory = v) ->
AX ((state = Available & inventory = v) |
    (state = Reserved & inventory = v-1)))
```

Note that the variable *inventory* partakes of the proposition added to this formula to verify this business rule. The variable *v* is used only to simplify the formula, since in SMV all the possible inventory values should be written.

Analogous to this example, there is other case: if the state is reserved and the service cancels the reservation, showed as follow:

```
AG ((state = Reserved in Auction & service = Cancel Auction &
   inventory = v) -> AX ((state = Available & inventory = v+1) |
   (state = Reserved & inventory = v)))
```

The next formulas illustrate some consistency properties of the English auction web site modeled.

The inventory should not be negative.

```
AG !(inventory < 0)
```

If the inventory is positive, at least one item must be available.

```
AG ((inventory > 0) -> (product_state = Available))
```

Finally some examples of isolation are presented.

If there are two buyer agents, one reserving the item and the other canceling his/her reservation, the inventory must be kept consistent after the execution of both services:

```
AG ((buyer1_service = Reserve in Auction & buyer2_service =
     Cancel Auction & inventory = v) -> AX (inventory = v))
```

In the case of *inventory = 0*, the reservation service can not be preceded by the cancellation service. So, to solve this problem we decide to give priority to the buyer agent that wants to cancel the reservation.

In a similar way, we specified all the other business rules and verify their veracity.

This version of the model has 7 modules (corresponding to 5 processes: item, buyer agent, seller agent, product, system), which corresponds to 226 lines of SMV code, and 30 properties verified.

4.4 Execution or Architectural Level

In this stage we added new modules to represent the e-commerce system as real as possible. So we include the web server, transaction server and database server in the model, adapting the specifications to it. Thus, the properties are related to requests, instead of services.

In this level we do not identify new properties related to business rules since all of them were verified in the previous levels. However, it was necessary to check the functioning of the architectural components, which demands the verification of new properties.

This version of the model has 8 modules (corresponding to 6 processes), which represent two buyer agents, a seller agent, the system (English auction server), the web server, the transaction server, two items(database server). The complete model demanded 693 lines of SMV code, and more than 70 properties were verified.

5 RELATED WORK

Model checkers have been successfully applied to the verification of several large complex systems such as an aircraft controller [6], a robotics controller [5], and a distributed heterogeneous real-time system [28]. The key to the efficiency of the algorithms is the use of *binary decision diagrams* [27] to represent the labeled state-transition graph and to verify if a timing property is true or not. Model checkers can exhaustively check the state space of systems with more than 10^{30} states in a few seconds [4, 6].

There are many works related to formal methods and more specifically to formal specification using symbolic model checking. But they often focus on hardware verification and protocols, rarely to software applications.

[12] describes the formal verification of SET(*Secure Electronic Transaction*) [30] protocol, proving five basic properties of its specification. The authors considered formal verification essential to demonstrate its correctness and robustness. In this work, they use the FDR verifier [29].

In [2] the authors presents a payment protocol model verification. Before this, we have knowledge of works in e-commerce only in authentication protocols. This article presents a methodology used to perform the verification, which is very interesting. They validate it using the *C-SET* protocol.

Formal analysis and verification of electronic commerce systems have not been studied in detail until recently. Most work such as [2, 12, 16, 32] concentrates on verifying properties of specific protocols and do not address how these techniques can assist in the design of new systems. Moreover, these techniques seem to be less efficient than ours, ranging from theorem proving techniques [2, 16] which are traditionally less efficient (even though more expressive), to model checking [12, 32]. But even these works tend to be able to verify only smaller systems consuming much higher resources than our method.

6 CONCLUSION AND FUTURE WORK

Bowen and Hinchey [3] present practical questions that invalidate myths related to formal methods and elaborate some conclusions that serve as motivation for our work:

- An important question is how to make easy the adoption of formal methods in software development process.
- Formal methods are not a panacea, they are an interesting approach that, as others, can help to develop correct systems.

This paper proposes a methodology to specify e-commerce systems. This technique can increase the efficiency of the design of electronic commerce applications. We use formal methods not only to formalize the specification of the system but also to automatically verify properties that it must satisfy. This technique can lead to more reliable, less expensive applications that might be developed significantly faster. We have modeled and verified an English auction web site in order to demonstrate how the method works.

The proposed method can be applied in general e-commerce systems, where the business rules can be modeled by state transitions of the items on sale. As the method is based on CTL-formulas, the business rules should be represented by them, what can be considered a limitation of the method.

We are currently studying other features of electronic commerce systems that we have not yet formalized, as well as the possibility of generating the actual code that will implement the system from its specification. In this context, we have been developing a set of design patterns to be used in the design and verification process of e-commerce systems. Based on the *Formal-CAFE* methodology, these patterns aim to simplify the adoption of this methodology. *Formal-CAFE* demands knowledge of symbolic model checking, which is considered a hurdle to its diffusion. The idea is to define a model checking pattern hierarchy, which specifies patterns to construct and verify the formal model of e-commerce systems. These patterns will be defined considering a *Formal-CAFE* case study of a virtual store.

We consider this research the first step to the development of a framework, which will integrate the methodology, an e-commerce specification language based on business rules, and a symbolic model checker. Another future work is to use our model checking patterns in other application areas, such as mobile e-commerce (m-commerce) and telecommunications.

References

[1] George S. Avrunin, Ugo A. Buy, James C. Corbett, Laura K. Dillon, and Jack C. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Transactions on Software Engineering*, 17(11):1204–1222, November 1991.

[2] Bolignano. Towards the formal verification of electronic commerce protocols. In *PCSFW: Proceedings of The 10th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1997.

[3] Jonathan P. Bowen and Michael G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(3):34–41, 1995.

[4] S. Campos, E. Clarke, W. Marrero, and M. Minea. Verifying the performance of the pci local bus using symbolic techniques. In *International Conference on Computer Design*, 1995.

[5] S. V. Campos, E. M. Clarke, W. Marrero, and M. Minea. Timing analysis of industrial real-time systems. In *Workshop on Industrial-strength Formal specification Techniques*, 1995.

[6] S. V. Campos, E. M. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing quantitative characteristics of finite-state real-time systems. In *IEEE Real-Time Systems Symposium*, 1994.

[7] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a reimplementation of smv, 1998.

[8] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: A new symbolic model verifier, 1999.

[9] A. Cimatti and M. Roveri. User manual.

[10] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[11] D. Rosenblum. Formal methods and testing: Why the state-of-the-art is not the state-of-the-practice. *ACM SIGSOFT Software Engineering Notes*, 21(4), 1996.

[12] Shiyong Lu Department. Model checking the secure electronic transaction (set) protocol. In *Proceedings of the 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 1998.

[13] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *2nd Workshop on Formal Methods in Software Practice*, March 1998.

[14] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *21st International Conference on Software Engineering*, May 1999.

[15] Matthew B. Dwyer and Lori A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 62–75, December 1994.

[16] Sigrid Gurgens, Javier Lopez, and Rene Peralta. Efficient detection of failure modes in electronic commerce protocols. In *DEXA Workshop*, pages 850–857, 1999.

[17] Zri Har'El and Robert Kurshan. Software for analytical development of communications protocols. *AT&T Technical Journal*, 69(1):45–59, January 1990.

[18] G. E. Hughes and M. J. Cresswell. *A Companion to Modal Logic*. Methuen, London, 1984.

[19] Wagner Meira Jr., Cristina Duarte Murta, Sérgio Vale Aguiar Campos, and Dorgival Olavo Guedes Neto. *Sistemas de Comercio Eletronico, Projeto e Desenvolvimento*. Campus, 2002.

[20] K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1992.

[21] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[22] P. Milgrom. The economics of competitive bidding: A selective survey. *In L. Horwicz, D. Schmeidler, and H. Sonneschein, editors*, 1985.

[23] P. Milgrom. Auctions and bidding: A primer. *Journal of Economic Perspectives*, 3:3–22, 1989.

[24] P. Milgrom and Robert Weber. A theory of auctions and competitive bidding. *Econometrica*, 50:1089–1122, oct 1982.

[25] A. Pereira, M. Song, G. Gorgulho, W. Meira Jr., and S. Campos. A formal methodology to specify e-commerce systems. In *Proceedings of the 4th International Conference on Formal Engineering Methods*, Lecture Notes in Computer Science, Shanghai, China, October 2002. Springer-Verlag.

[26] R. Cleaveland and J. Parrow and B. Steffen. The concurrency workbench: A semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, Jan. 1993.

[27] R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[28] S. V. Campos. *A Quantitative Approach to the Formal Verification of Real-Time Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1996.

[29] F. Systems. Fdr: A tool for checking the failures-divergence preorder of csp, 1999.

[30] M. Visa. Secure electronic transaction (set) specification book 1,2,3, 1997.

[31] W. Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *Journal of Finance*, 16:8–37, mar 1961.

[32] W. Wang, Z. Hidvégi, A. Bailey, and A. Whinston. E-process design and assurance using model checking. In *IEEE Computer*, Oct. 2000.

View publication stats