Adriano César Machado Pereira

Uma Metodologia para Verificação de Modelos de Sistemas de Comércio Eletrônico

(A Model Checking Methodology for E-commerce Systems)

Dissertação apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Belo Horizonte

28 de Agosto de 2002

UNIVERSIDADE FEDERAL DE MINAS GERAIS

FOLHA DE APROVAÇÃO

Uma Metodologia para Verificação de Modelos de Sistemas de Comércio Eletrônico

ADRIANO CÉSAR MACHADO PEREIRA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]

Prof. Wagner Meira Júnior – Orientador
Departamento de Ciência da Computação – ICEx – UFMG

[illegible]

Prof. Sérgio Vale Aguiar Campos – Co-orientador
Departamento de Ciência da Computação – ICEx – UFMG

[illegible]

Prof. Carlos José Pereira de Lucena
Departamento de Informática – PUC-Rio

[illegible]

Prof. Virgílio Augusto Fernandes Almeida
Departamento de Ciência da Computação – ICEx – UFMG

Belo Horizonte, 28 de agosto de 2002.

Resumo

Comércio eletrônico é uma importante área de aplicação associada à computação que tem evoluído significativamente nos últimos anos. Entretanto, sistemas de comércio eletrônico são complexos e difíceis de serem projetados corretamente. Atualmente, a maioria das abordagens é *ad-hoc*, o que normalmente torna os sistemas menos confiáveis e implica em alto custo em termos de tempo e recursos. Além disso, garantir a corretude de um sistema de comércio eletrônico não é tarefa trivial devido à grande variedade de erros, muitos deles muito sutis. Tal tarefa é complexa e trabalhosa se apenas testes e simulação, técnicas comuns de validação de sistemas, são utilizados. Neste trabalho propõe-se uma metodologia que utiliza métodos formais, especificamente verificação simbólica de modelos, para projetar aplicações de comércio eletrônico e verificar automaticamente se suas regras de negócio são satisfeitas. Usando a metodologia proposta, o projetista é capaz de identificar, antecipadamente, erros no processo de desenvolvimento do projeto e corrigí-los antes que se propaguem a estágios posteriores da implementação. Dessa forma, torna-se possível gerar aplicações mais confiáveis, desenvolvidas mais rapidamente e a baixo custo. A fim de demonstrar a aplicabilidade e a praticabilidade da técnica proposta, modelou-se e verificou-se uma loja virtual, na qual múltiplos compradores competem para adquirir itens de um produto. A utilização de verificação automática se mostrou de extrema importância, pois indicou erros difíceis de serem detectados durante o projeto da aplicação como, por exemplo, uma falha do controle de concorrência que permitia que o mesmo artigo fosse vendido para clientes distintos. A metodologia proposta pode ser aplicada em sistemas de comércio eletrônico em geral, onde as regras de negócio podem ser modeladas através de transições no estado dos itens a venda. Como o método proposto é baseado em fórmulas CTL, as regras de negócio devem ser representadas através das mesmas, o que pode ser considerado uma limitação da metodologia. Estamos estudando outras características dos sistemas de comércio eletrônico que ainda não foram formalizadas, assim como a possibilidade de geração do atual código que implementa o sistema a partir de sua especificação. Neste contexto, estamos desenvolvendo um conjunto de padrões de projeto a serem utilizados no projeto e verificação de sistemas de comércio eletrônico. A idéia principal é definir uma hierarquia de padrões para verificação de modelos, que especifique padrões para construção e verificação de modelos formais de comércio eletrônico. Consideramos esse trabalho o primeiro passo para o desenvolvimento de um ambiente que integre a metodologia, uma linguagem de especificação de sistemas de comércio eletrônico baseada em regras de negócio, e um verificador de modelos. Um trabalho futuro é aplicar a metodologia proposta em outras áreas, tais como comércio eletrônico móvel e telecomunicações.

Abstract

Electronic commerce is an important application that has evolved significantly in recent past. However, electronic commerce systems are complex and difficult to be designed correctly. Current most approaches are *ad-hoc*, frequently leading to expensive and 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 subtle errors may occur. Such task is quite hard and laborious if only tests and simulation, common techniques for system validation, are used. In this work we propose 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 a virtual store in which multiple buyers compete for product items. For instance, the verification process pointed out a concurrency control error which allowed the same item to be sold twice. 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 been 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 for 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.

Agradecimentos

Agradecimento significa ato ou efeito de agradecer, que significa demonstrar ou manifestar gratidão, que denota um reconhecimento por um benefício recebido. Imagino que normalmente o agradecimento deva ser curto, mas penso que não posso perder esta oportunidade para expressar minha eterna gratidão.

"Eu guardo, para todos aqueles que de uma ou outra forma contribuíram para fazerme mais grata a vida, uma eterna gratidão, e estampo nessa gratidão a lealdade com que conservo essa recordação, a qual jamais pôde empalidecer ali onde se encerra tudo quanto constitui a história de minha vida." (RAUMSOL)

A gratidão é um dos mais nobres sentimentos do ser humano. Penso que ela engrandece a quem a testemunha e estimula e faz feliz a quem a recebe. Porém a gratidão é um sentimento que vai ficando mais raro à medida que o ser humano, invertendo a hierarquia de valores, tem se envaidecido, olvidando a Fonte Suprema que o criou, as suas raízes espirituais e os seus deveres para com o seu semelhante.

Gratidão rima com recordação, e me recordo de um sábio ensinamento que expressa que "recordar o bem recebido é fazer-se merecedor de tudo quanto amanhã possa nos ser brindado".

A conclusão desta etapa, deste trabalho é um marco importante para mim. Retrata um esforço não só meu, mas de muitos, com quem tenho a alegria de poder compartilhar este momento especial.

Inicialmente, gostaria de expressar minha gratidão a Deus, pela maior de todas as oportunidades, que é a de viver, por presidir sempre todos os momentos de minha vida.

"Que sempre seja Deus quem presida suas horas de alegria, oferecendo-Lhe, do mais íntimo do coração, sua gratidão por tudo o que Lhe deve e possui em felicidade, em conhecimento, em conforto, em triunfos."

Vale dizer também que "é bom recordar o que mais de uma vez afirmamos: os seres invocam a Deus em seus momentos de desventura, pretendendo um amparo imediato, sem perceberem, por outro lado, que poucos o fazem como homenagem de gratidão por seus momentos de felicidade e, menos ainda, para mostrarem-lhe o fruto de seus esforços por vincular-se à sua maravilhosa Vontade, plasmada na Criação. É preciso, pois, recordá-lo também nos momentos de alegria; a recordação não só perde assim o caráter especulativo, como também brota da gratidão pela felicidade que se vive. Então sim, o espírito individual pode elevar a alma e vinculá-la a vibrações superiores."

Sou eternamente grato as Américas, avós amáveis e companheiras, ao avô Zizico, brincalhão e paciente, avô Wilson, sinônimo de luta e responsabilidade.

A meus queridos pais, Elza e Lourival, pelo ilimitado afeto e verdadeiro amor. Agradeço o apoio, o carinho e os ensinamentos de inestimável valor. A Lê e Déia, por sempre terem me dado tanto apoio e amor. Não posso deixar de lembrar do Eduardo, Igor e todos os tios, tias, primos e primas, que completam esta minha maravilhosa família. Merecem minha recordação também Marcinha, Maísa, Marly e Marcos.

A Maira, minha Lindona, com quem compartilho os mais valiosos anelos, por seu amor incondicional, afeto grandioso, respeito, companherismo, tolerância e paciência. Agradeço também por saber respeitar meu trabalho, me auxiliar nas decisões e compreender minhas ausências. Obrigado por estar tornando minha vida mais feliz ainda!

Ao professor e Amigo Wagner Meira Jr., orientador deste trabalho e de tudo quanto realizei nos últimos 5 anos de pesquisa, agradeço pelos valiosos ensinamentos, pelas oportunidades, desafios, orientações e incentivos. Você costuma dizer que "o que vale no curso de mestrado ou doutorado é o sofrimento". No início, não entendia isso, hoje compreendo que na verdade esse sofrimento a que você se refere é a luta que, com muito esforço e perseverança, culmina em capacitação e conhecimento. Sou muito grato por sua confiança e colaboração.

Também agradeço ao professor Sérgio Campos, co-orientador deste trabalho, pelos conhecimentos transmitidos e importante colaboração.

Aos demais membros da banca, os professores Virgílio Augusto Fernandes Almeida e Carlos José Pereira de Lucena, ilustres pesquisadores de reconhecimento internacional, agradeço pelos generosos conhecimentos e conselhos.

Agradeço aos companheiros de pesquisa Mark Song e Gustavo Gorgulho pelas inúmeras contribuições a este trabalho, onde juntos formamos uma pequena grande equipe e alcançamos méritos com muito esforço, dedicação e perseverança. Espero que nossa união e trabalho renda ainda muitos frutos e que os frutos já colhidos possam ser úteis para muitos.

Aos amigos, que felizmente são muitos, tantos que não caberiam nesta seção do trabalho. Em especial, aos amigos de infância, do Colégio Logosófico e Fundação Logosófica, onde destaco Alexandre Silveira, Ana Carolina Chaves, Anita Lima, Cynthia Porto, Daniel Rocha, Guilherme Cavalieri, Gustavo França, Sarah Angrisano. Aos amigos do grupo *OGDGA*, onde destaco Luciana Avelar e Rodrigo, o Foca. Aos integrantes das minhas famílias de segundo grau, das quais destaco a família Neves Cavalieri, França, Silva Gontijo, Macedo Paulinelli, e Alves Pereira. Aos amigos da *Smart Price*, incluindo os que lá estão e os que por lá já passaram. A todos os amigos da *2000 e-mail*. A todos os amigos da UFMG, onde destaco, sem ordem de relevância mas com diferentes graus de afeto, Adriano Veloso, Ana Paula Atayde, Autran Macedo, Benício Gontijo, Bruno Diniz, Bruno Pôssas, Eduardo Kraemer, Eveline Veloso, Fabiana Ruas, Fernando Caixeta, Flávia Ribeiro, Goedson Paixão, Gustavo Gama, Júnia Gaudereto, Patrícia Marques, Paulo Araújo, Rodrigo Barra, Rodrigo Fonseca, Tanara Lauschner, Tassni, Wolber, entre outros. A todos os amigos da Engenharia, em especial ao José Henrique e Leonardo Santos. Prestar um agradecimento nominal não me parece a forma mais generosa de manifestar meu agradecimento, pois o verdadeiro reconhecimento é visível no afeto que existe entre cada um de nós.

A todos do Laboratório *e-SPEED*, do Departamento de Ciência da Computação da UFMG, por tornarem alegre, rico e divertido nosso convívio diário.

Não poderia deixar de agradecer ao Departamento de Ciência da Computação (DCC) da Universidade Federal de Minas Gerais (UFMG) pelo ambiente propício ao desenvolvimento do trabalho, muitos professores capacitados e competentes funcionários de que dispõe. Agradeço especialmente à Emília, Luciana, Renata e Túlia, funcionárias do DCC, que com grande eficiência deram suporte ao meu curso de pós-graduação.

Finalmente, dedico este trabalho a todos que não estão diretamente mencionados, mas que de alguma forma colaboraram para a conclusão do mesmo. Meu sincero obrigado! Espero que este trabalho possa ser fonte de conhecimento para muitos seres e base para desenvolvimento de novas pesquisas.

Agora que estou a finalizar estes dizeres, noto que palavras não são suficientes para expressar meu afeto e gratidão, meu verdadeiro sentir.

Para concluir recorro então a um ensinamento especial de um grande Mestre, a quem tanto sou grato, pois tem verdadeiramente tornado minha vida muito mais feliz: "A vida não deve terminar como terminam as horas do dia, agonizando em um entardecer. A vida tem que ampliar seus horizontes; fazer longas as horas da existência para que o espírito, incorporado na matéria, experimente a grandiosidade de sua criação. Para isso tem que renovar-se no passado e no futuro. No passado, reproduzindo constantemente na tela mental todas as passagens vividas com maior intensidade; no futuro, pensando no que ainda resta por fazer, naquilo que se pensou fazer, e, sobretudo, no que se quer ser nesse futuro. E quanto mais gratidão o homem experimente pelo passado, quanto mais gratidão guarde pelas horas felizes vividas nele, assim como pelas de luta ou de dor, que sempre são instrutivas, tanto mais abrirá sua vida a novas e maiores perspectivas de realização."

Contents

List of Figures

List of Tables

Chapter 1

Introduction

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 define electronic commerce as the use of 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 system. 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 cause damages to the users and to 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 simulations, 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 since 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 for the sake of modeling, formal specification and formal verification of the system, that are the basic processes of 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.

The next sections describe important concepts about formal methods, temporal logic and model checking, and e-commerce systems. Then, the objective and main contributions of this master thesis are presented. After that, there are two sections explaining the related work and organization of the document, respectively.

1.1 Formal Methods

Formal methods are a set of techniques that use mathematical notation to describe the requirements of the system and detail forms to validate this specification and subsequent implementation.

The term formal methods refers to the use of mathematical modeling, calculation, and prediction in the specification, design, analysis, construction, and assurance of computer systems and software. The reason it is called "formal methods" rather than "mathematical modeling of software" is to highlight the nature of the mathematics involved.

The specification corresponds to one of the initial stages of software development process [40, 43] and its objective is to define, in a complete and consistent way, the functional requirements of the system. In general, the specification is written in natural language, being subjected to ambiguities, sensitivity to the context and different interpretations. Formal specification consists of the use of formal notations, based in mathematical techniques and formal logic, to specify systems. The use of formal notations and mathematical formalisms allows to reduce errors and ambiguities committed during this process, generating an accuracy and not ambiguous specification.

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 and promising formal method technique to verify hardware and software systems using temporal logic.

1.2 Temporal Logic and Model Checking

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 [51].

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.

In this work we use *Symbolic model checking*, which is a formal verification approach by which a desired behavioral property of a system can be verified over a model through representation 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 some temporal logic.

CTL (Computation Tree Logic) is a language used to describe properties of systems that will be verified by the model checker. *Computation trees* are derived from state transition graphs. 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.

The model checking system we use is called SMV [59], developed by McMillan as part of his doctoral dissertation thesis. It is based on a language for describing hierarchical finite-state concurrent systems. SMV is a tool for checking finite state systems against specifications in the temporal logic CTL. These concepts are very important to help us in the conception of the methodology to design e-commerce systems, which are presented in the next subsection.

1.3 E-commerce Systems

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.

Generically, we define commerce as the execution of commercial transactions involving trade goods and services. E-commerce consists of a set of techniques and computer technologies used to support commerce or make it easier. An English auction web site and a virtual store are traditional examples of e-commerce applications.

Electronic commerce servers and traditional WWW servers act in well distinct contexts [25]. Furthermore, there are many differences between these types of servers. However, they can be differentiated by the additional functionalities supported and the information stored by the e-commerce servers. WWW servers only receive and answer requests, while e-commerce servers keep information transmitted between the user and the server, and their associated actions. This information is kept for the purpose of transactional integrity and support to the offered services. The state of the server comprises the user session, which represents all the interactions that a user makes with the site in one sitting. The following subsections describe the architecture of an e-commerce server.

1.3.1 Architecture

Components

An electronic commerce server [77] can be divided into three integrated components (Figure 1.1):

1. **WWW Server**: it is the manager of the tasks, being responsible for the interface with the users (customers), interacting directly with them, receiving the requests, sending them to transaction server and repassing the results. It provides the interface between the client access tool (normally a *browser*) and the e-commerce server.

2. **Transaction Server**: It processes the requests submitted to the e-commerce server, such as the addition/removal of a new product to the shopping cart.

3. **Database**: It stores all the information of the virtual store, such as the description of the product and the level of supply. More than a simple repository, it adds several functionalities that allow the standardized, safe, and efficient access to the data, through, for example, the creation of an index and user access control.

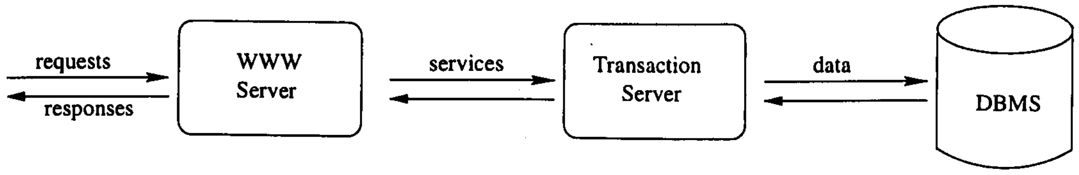

Figure 1.1: Three-level architecture of e-commerce server

Requirements

There are four essential requirements to the implementation of electronic commerce servers [68, 76]:

1. **Management of the state of the application**: The state of the application is the set of user information and its interactions while accessing the server. Particularly, the management of the state of the application makes possible the authentication of users, control of user session, and the use of personalized services.

2. **Transactional Support**: These requirements are related to the transactions that satisfy certain characteristics traditionally grouped into four properties under the acronym ACID (Atomicity, Consistency, Isolation, Durability). These features make possible, among others, concurrence control on the database and mechanisms of recovery in case of errors.

3. **Security**: The security requirements are related to the restrictions of access to the data managed by the server. The *Web* is not really a safe environment and the execution of e-commerce applications must take in consideration the basic requirements of access restrictions to objects of the data base.

4. **Performance**: The performance of e-commerce servers is a crucial factor for the satisfaction of the customers and consequent accomplishment of commercial transactions.

1.3.2 Properties

In an e-commerce server, focusing more specifically at the transaction server, a transaction can be understood as a sequence of operations executed on the existing products in the site capable to cause changes in the product state, which is part of the e-commerce application state. Despite this, it is important to know the concept of transaction atomicity. Atomic transactions are constantly used in distributed or concurrent systems, being characterized by the fact that, if something hinders the end of this transaction, the state of the application must be restored to the state before the beginning of the execution, preventing that the system pass to an inconsistent state.

The study of our model identified three relevant classes of properties to be verified:

- Atomicity: a transaction must be executed in its totality. If it is not possible, does not have to execute none of its operations.
- Consistency: transformations preserve the consistency of the product state, that is, a transaction transforms a consistent product state into another consistent one, without necessarily preserve the consistency in the intermediate points of the transaction.
- Isolation: the transactions executed in the virtual store that modify the product state must be isolated one of the others, thus, a transaction can not affect the result of other executed simultaneously.

1.4 Objective

The objective of this work is to propose a methodology to design e-commerce systems with model checking support. It can be divided into four main issues. The first one comprises the study and evaluation of the available methodologies to specify e-commerce systems. The second one is to study formal methods, more specifically *model checking*, to define which technique must be adopted. The third one is to create the methodology that uses formal-method techniques to design electronic commerce systems and to automatically verify that these designs satisfy important properties of these systems. The fourth one is to validate the methodology using traditional examples of e-commerce systems.

1.5 Contributions

Through our work it has become evident that there are few methodologies to design e-commerce systems and none of them apply model checking.

The main contributions of this work are:

- Develop a work that involve three important areas of computer science: electronic commerce, model checking, and software engineering.

- Provide a methodology to design more reliable e-commerce systems.
- Opportunity to extend the work to other application areas.

1.6 Related Work

Model checkers have been successfully applied to the verification of several large complex systems such as an aircraft controller [16], a robotics controller [15], and a distributed heterogeneous real-time system [91]. The key to the efficiency of the algorithms is the use of *binary decision diagrams* [85] 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 [14, 16].

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.

The work described in [9] presents practical questions that invalidate myths related to formal methods, explained in Appendix B, 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.

[29] describes the formal verification of SET(*Secure Electronic Transaction*) [94] 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 [93].

According to [8], there is much interest in improving embedded system functionalities, where security is a critical factor. The use of softwares in this systems enable new functionalities, but create new possibilities of errors. In this context, formal methods might be good alternatives to avoid them. But the authors mentioned that formal methods are rarely adopted in this system, where security is a basic requirement.

In [7] 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 [7, 29, 44, 96] 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 [7, 44] which are traditionally less efficient (even though more expressive), to model checking [29, 96]. But even these works tend to be able to verify only smaller systems consuming much higher resources than our method.

1.7 Organization

This master thesis introduces a methodology to create robust and correct e-commerce systems applying model checking. Chapter 2 explains important concepts of e-commerce modeling and explains our methodology, *Formal-CAFE*. Chapter 3 shows two case studies used to validate the *Formal-CAFE* methodology. Chapter 4 presents some conclusions and future work. For sake of completeness some appendices are also presented: Appendix A describes the *CAFE* methodology; Appendix B discusses in details the concepts of formal methods; Appendix C shows an overview of model checking; finally Appendix D introduces the SMV and NuSMV systems, explaining important features about them.

Chapter 2

Model Checking E-commerce Systems

There are many types of e-commerce applications, such as virtual bookstore, auction sites, and others. The difference between them are their nature and their business rules. Some business rules are common, for example: a given 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.

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.

As showed in [78, 79], properties can be described as formulas in CTL, which are built from atomic propositions, boolean connectives, and temporal operators.

Consider the following example: 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)) → 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.*

In [32, 33] was developed a system of property specification patterns for finite-state verification tools based in the scope, order and occurrence of an event[1]. The next subsection explains this property specification patterns.

2.2 Property Specification Patterns

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 [26], 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 [33]. 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 [58], bisimulation [80], language containment [47], flow analysis [34], and inequality necessary conditions [5]. 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 [33], 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.

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

So we think this property specification patterns can ease the adoption of model checking. Each property specification pattern has a scope, which is the extent of the program execution over which the pattern must hold [32, 33]. There are five basic kinds of scopes, as we explain follow:

- Global: the entire program execution
- Before: the execution up to a given state / event
- After: the execution after a given state / event
- Between: any part of the execution from one given state / event to another given state / event
- After-until: like between but the designated part of the execution continues even if the second state / 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 $Q_1, ..., Q_n$.

The complete hierarchy created by [33] is showed in Figure 2.1.

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

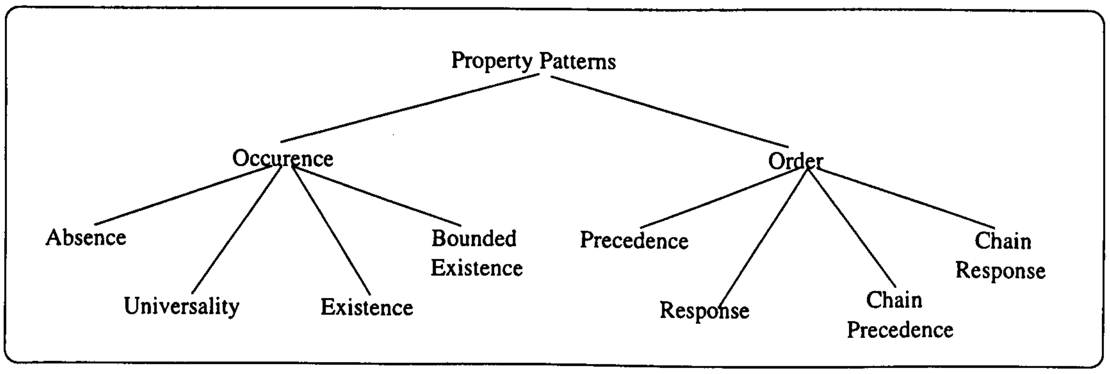

Figure 2.1: A Pattern Hierarchy

To describe a portion of a system's execution that is free of certain events we present, as an example, the *Absence Pattern*, also known as *Never*. In CTL Never(P) can be mapped as:

- Globally: AG (!P).
- Before R: A [!P U (R $\vee$ AG(! R))] .
- After Q: AG (Q $\rightarrow$ AG (! P)).
- Between Q and R: AG (Q $\rightarrow$ A [!P U (R $\vee$ AG (!R))]).
- After Q until R: AG (Q $\rightarrow$!E [!R U (P $\wedge$!R)])

In the next subsection we introduce the formal model of e-commerce systems defined by our methodology, using examples to demonstrate how it can be adopted.

2.3 Formal model 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 [78, 79].

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's manager. The agents perform actions that may change the state of an item, that is, actions correspond to transitions in the life cycle graph. Putting an item in the basket or canceling 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.

In the next subsection we explain the relevant properties to verify in e-commerce systems.

2.3.1 Properties of an e-commerce 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. Note that this property is not inherent to the application business rules. So it should be verified in the first stage, before verifying the other properties related to business rules.

To express this property we can use the property pattern of *Existence*. Additionally, it's necessary to define the scope as *after*, considering that "Exists in the Future" means "after current state / event". Once the completeness is verified true, we identify other relevant properties to be checked, related to the business rules.

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

Most properties to be verified in the virtual store 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 verify whether transactions are atomic.

We have to verify three types of properties that relate to the consistency of transactions:

- Atomicity: A transaction must be finished or not started, that is, if it doesn't finish, its effects have to be undone.

- Consistency: A transaction transforms a consistent state into another consistent one, without necessarily preserving the consistency in the intermediate points of the transaction. 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.

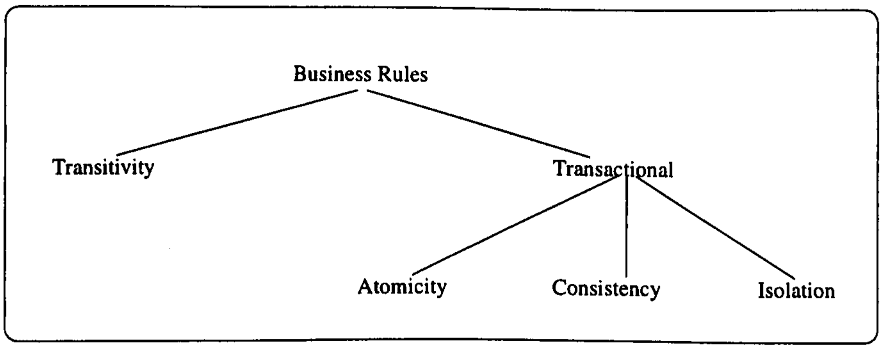

Figure 2.2: Classification of business rules

The Figure 2.2 illustrates our classification of business rules. The next section details the levels of the formal methodology, using real examples of e-commerce business rules to explain how this properties can be checked.

2.4 Formal-CAFE: An Incremental Specification Methodology with Formal Verification

This section explains the methodology to design the verifiable e-commerce system incrementally. Our methodology is hierarchical and divided into four major levels. Before explain the levels is primordial to dissert how to describe an verifiable e-commerce system properly.

The proposed methodology, an extension of the *CAFE* methodology (Chapter A), consists 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 [58], to build the model.

The methodology is incremental and divided into four major levels. It is relevant to emphasize that this 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.

The next following subsections describe in detail each level of our methodology.

2.4.1 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 $D_i = D_j$.

In this first level we don't 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 make easy the work being developed in the next levels of the methodology.

2.4.2 Application Level

This level describes the e-commerce system in terms of the life cycle of the items. It's 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.

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 2.3. States in this graph are possible states for the item such as *available*, or

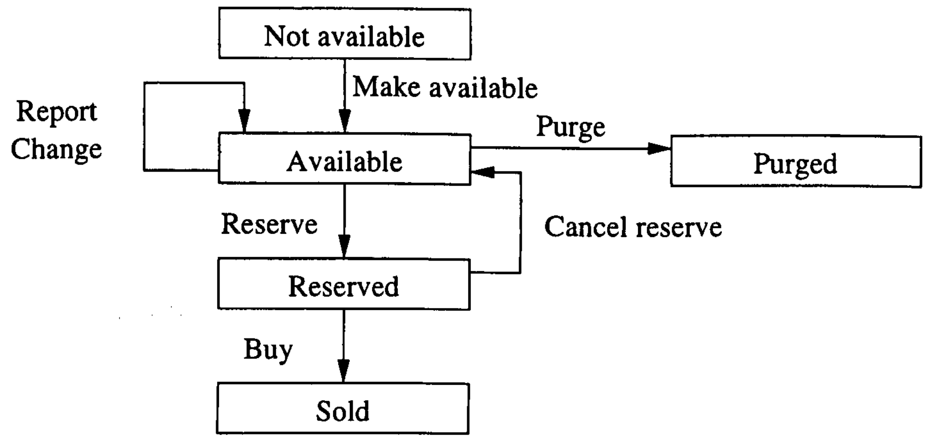

Figure 2.3: The life cycle graph of product's item

reserved. Transitions represent the effect of actions such as reserving an item or buying it.

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.

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 level it's important to verify the completeness property of the e-commerce model. Here, it's 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 2.4 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.

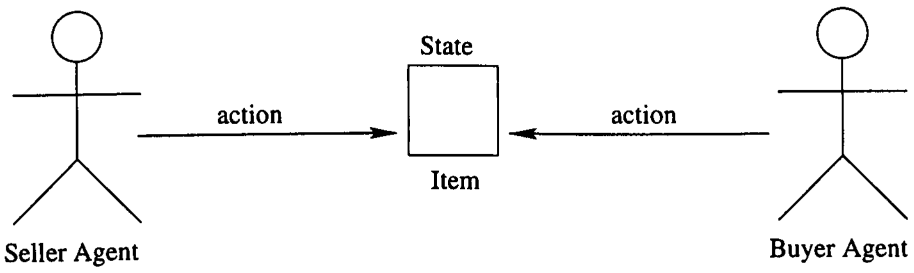

Figure 2.4: The First Level of the Methodology

2.4.3 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, that we called 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 services are executed and what happens with the product and its items in this case.

In this level it's 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 ((state = Not available & service = Make available) ->
   AX (state = Available)
```

or

```
AG ((state = Available & service = Purge) ->
   AX (state = Purged)
```

In this level it's important to verify either the atomicity, consistency and isolation properties either. It's essential to check the consistency between the state of the product and its items in a given moment. In this level, 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 action is performed by a buyer and granted by the server, the item must be reserved in the next state and the inventory must be decremented.

```
AG ((state = available & service = reserve & inventory = 1) ->
   AX (state = reserved & inventory = 0))
```

Examples of consistency properties can be seen below:

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

  ```
  AG (pr1.inventory = 0 -> !pr1.available)
  ```

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

  ```
  AG (pr1.inventory > 0 -> pr1.available)
  ```

  or

  ```
  AG (pr1.inventory > 0 ->
     (it1.state = available) | (it2.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 ((ba1.service=reserve & ba2.service=reserve & inventory = 2) ->
  AX inventory = 0)
```

This level is depicted in Figure 2.5. 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.

It's important to notice that the properties validated in the first level should retain their validity in the second 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.

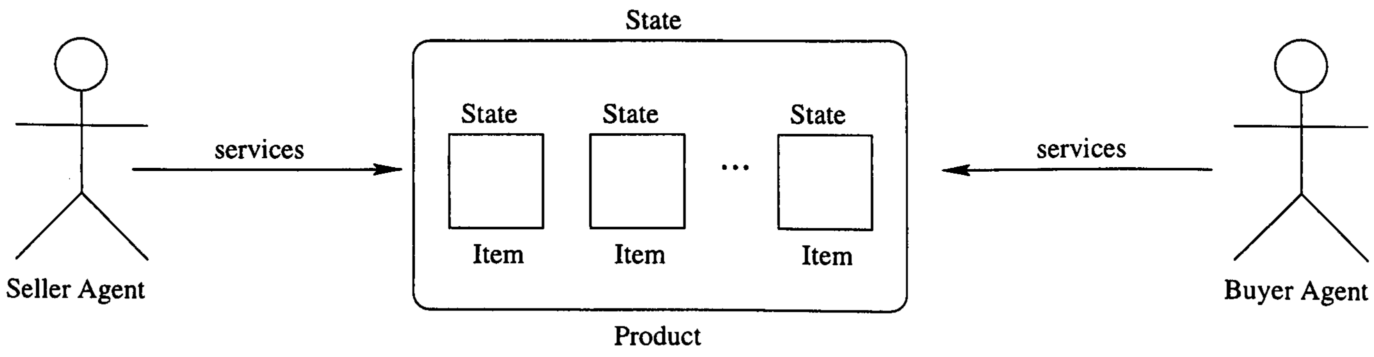

Figure 2.5: The Second Level of the Methodology

2.4.4 Execution or Architectural Level

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

In this level the model is more complex, contemplating the components of the system's architecture. Therefore the transactional properties, as well as transitivity and completeness properties, should be checked again to guarantee that the properties previously verified continue to be satisfied in the current version.

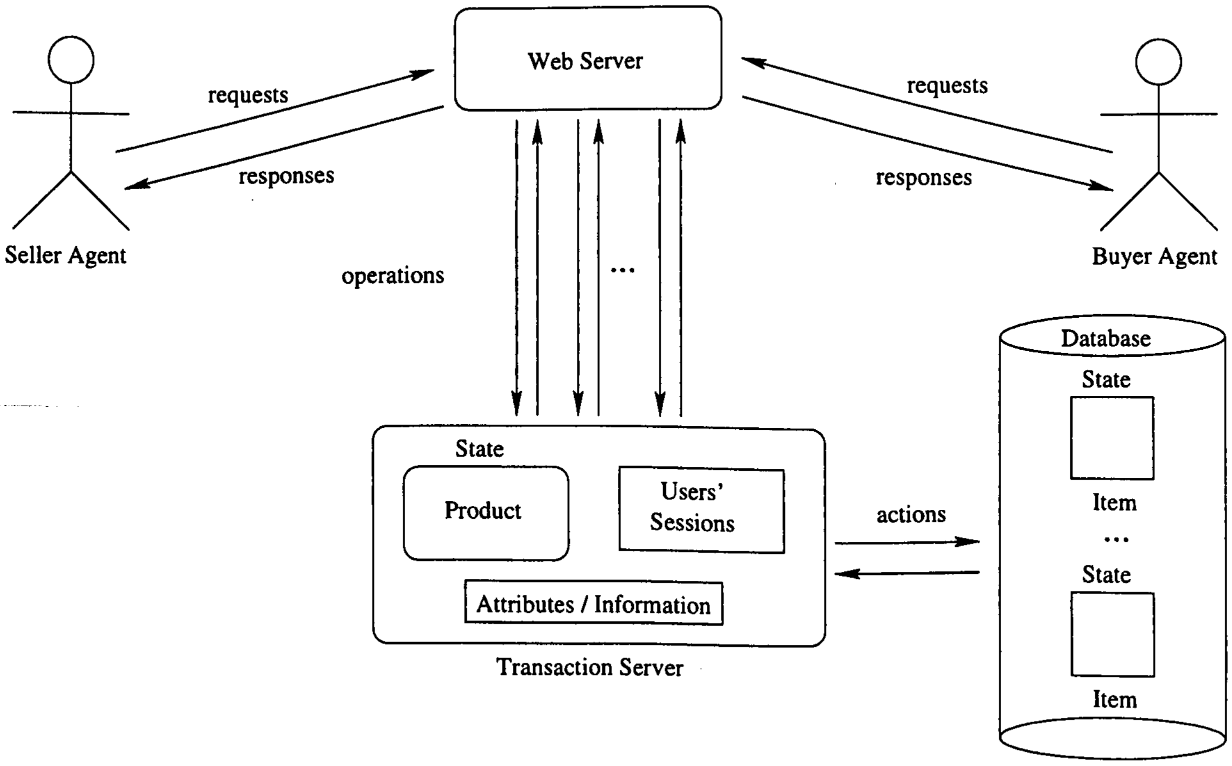

Figure 2.6: The Third Level of the Methodology

As shown in Figure 2.6, we introduce the components of the system: the transaction server of the virtual store and the web server. There are agents that submit requests to the web server, which transform 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. The next chapter presents case studies that validate the proposed methodology.

Chapter 3

Case Studies

3.1 Case Study: An E-commerce Virtual Store

In this section, we present a case study, an e-commerce virtual store, a very useful and popular application. The objective is to show how the methodology proposed can be used to design more reliable e-commerce systems. This is a typical electronic commerce 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 [17, 18, 19] to perform this task.

3.1.1 Conceptual Level

In the virtual store we modeled, there are six states which correspond to types of pages on the web site: *Home*, *Browse*, *Search*, *Select*, *Add* and *Pay*. The *Home* page is the initial web page of the site. The *Select* page shows specific information about a product. The *Add* page confirms product reservations and displays the contents of the customer's shop cart. The *Pay* page is loaded after the purchase of the items in the shop cart is completed. The *Search* and *Browse* pages present general information about the products offered by the virtual store. There are still some states that correspond to the administration view of the web site, used by the seller agent to change information about the products (operation *Change*) and modify the its inventory (operation *Make Available*).

The transitions between these pages are associated with actions executed by the agents. An example is the execution of a reserve action by the buyer agent, that causes the transition from *Select* to *Add* if completed with success. Therefore a transition between two web pages is mapped to an action in the life cycle graph of the product's item.

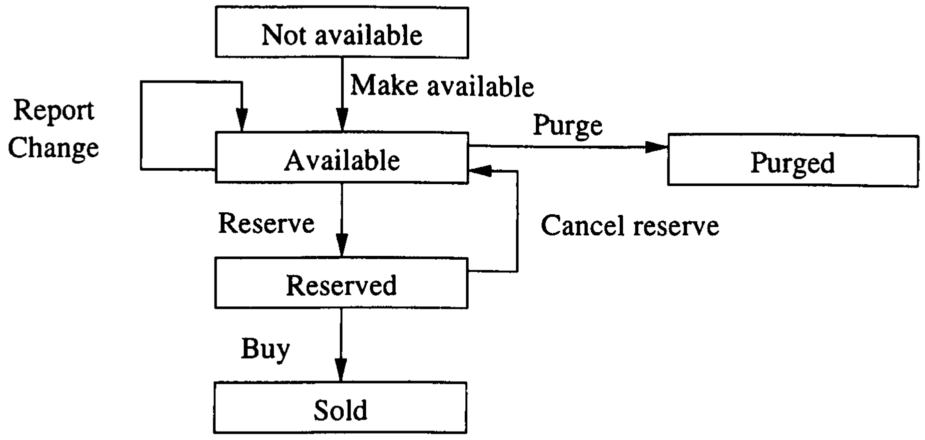

Figure 3.1: The life cycle graph of product's item

In our virtual store, we have modeled two types of agents. The *Buyer* Agent represents the customers that access the virtual store through WWW to get information about the product and potentially to buy it. The *Seller* Agent represents the product's supplier that will make it available, update its data and potentially sell it. The buyer agent can execute one of the following actions: *Report*, the client requests information about the product; *Reserve/Cancel Reserve*, the client reserves an item or cancel a previous reserve; and *Buy*. The seller agent can execute one of the actions: *Make Available*, when a new item enters the store; *Change* to change its attributes; and *Purge*, when the item is removed from the store.

We will model only one product in the store. This is because in this store transactions on different products do not interfere with one another and there are no properties that relate more than one product. As a consequence, modeling different products yields the same results for each one. This is not a restriction of the method, however, if the system relates different products, they could be modeled in a straightforward way. We do model multiple items for the product, because transactions may affect more than one item.

The life cycle graph of the product's item has the following states, as can be seen in Figure 3.1: *Not Available*, *Available*, *Reserved*, *Sold* and *Purged*. The transitions in the graph can be seen in the figure. The global model of the virtual store 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. For example, if a reserve is requested for one item but several are available, the store must decide which item will be reserved. An special array of variables keeps track of which items are available, and requests are fulfilled by an item chosen non-deterministically.

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 virtual store.

As defined in the methodology, Section 2.4.1, conceptual level details the e-commerce system requirements.

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* occurs, the next state is *Reserved.*
- If the item is in the state *Reserved* and the action *Cancel Reserve* occurs, the next state is *Available.*
- If the item is in the state *Reserved* and the action *Buy* occurs, the next state is *Sold.*
- 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* and *Cancel Reserve* must be atomic.
- If there are agents executing concurrently, their actions 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 virtual store.

3.1.2 Application Level

To design this level, it's important to know some concepts related to SMV. In SMV, variables are of type Boolean, integer or enumerated. The variable state is an enumerated variable that has five possible values. Assignment of values to variables is accomplished using the `init` and `next` commands. `init` defines the initial value of the variable and `next` defines the value of the variable in the next state as a function of the values of variables in the current state.

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 ba1: buyer_agent();
    ba2: buyer_agent();
    sa1: seller_agent();
    it1: item();
```

As described in Section 2.4.2, 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)
EF (it1.state = Sold)
EF (it1.state = Purged)

EF (ba1.action = Report)
EF (ba2.action = Report)
EF (ba1.action = Reserve)
EF (ba2.action = Reserve)
EF (ba1.action = Cancel_Reserve)
EF (ba2.action = Cancel_Reserve)
EF (ba1.action = Buy)
EF (ba2.action = Buy)

EF (sa1.action = Make_Available)
EF (sa1.action = Change)
EF (sa1.action = Purge)
```

These specifications should be consistent with the item's life cycle graph, as illustrated by Figure 3.1. In our model all of them were verified as *true*, certifying its completeness.

This version of the model has 3 modules (corresponding to 3 processes), which corresponds to 94 lines of SMV code, and 14 properties verified.

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

3.1.3 Functional Level

Continuing the process defined 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 2.4.3, we have to check the transitivity properties of the model. We can perform this using the following CTL formulas:

```
AG(state = Not_Available & service = Make_Available) ->
   AX(state = Available)
AG(state = Available & service = Purge) -> AX(state = Purged)
AG(state = Available & service = Change) -> AX(state = Available)
AG(state = Available & service = Report ) -> AX(state = Available)
AG(state = Available & service = Reserve) -> AX(state = Reserved)
AG(state = Reserved & service = Cancel_Reserve) ->
   AX(state = Available)
AG(state = Reserved & service = Buy) -> AX(state = Sold)
```

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 & 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 & service = Cancel_Reserve & inventory = v) ->
AX ((state = Available & inventory = v+1) |
(state = Reserved & inventory = v)))
```

The next formulas illustrate some consistency properties of the virtual store 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 & buyer2_service = Cancel_Reserve &
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.

As a result of our 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 following code fragment illustrates a buyer user session. Each buyer has its own user session.

```
next(user_session) := case
    ...
    user_session = product_select & buyer_service =
    Reserve & next(productIsAvailable)=1 : add_to_cart;
    ...
    user_session = product_select & buyer_service =
    Reserve & next(productIsAvailable)=0 : error;
    ...
    1: user_session;
esac;
```

In this situation, the transaction server allowed both clients to reserve the same item and the virtual store reached an inconsistent state. To solve this problem we introduced a semaphore to guarantee mutual exclusion in add to cart mechanism, as showed in the code fragment:

```
next(in_use) := case
    in_use=0 & buyer_service_1=Reserve & buyer_service_2=Reserve
& inventory > 0 & inventory <= 2: {1,2};
    ...
    in_use = 1 & buyer_service_1 = Cancel_Reserve : 0;
    in_use = 1 & buyer_service_1 = Buy : 0;
    in_use = 2 & buyer_service_2 = Cancel_Reserve : 0;
    in_use = 2 & buyer_service_2 = Buy : 0;
    1: in_use;
esac;
```

In such case, when a buyer requests a reserve of an item, the item will be added to its shop cart only if the variable *in_use* contains this buyer identification number.

This version of the model has 4 modules (corresponding to 5 processes), which corresponds to 211 lines of SMV code, and 30 properties verified.

3.1.4 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 level. 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 5 modules (corresponding to 7 processes), which corresponds to 700 lines of SMV code, and 71 properties verified.

The complete model has more than 10^{23} states with more than 10^{14} reachable states.

3.2 Case Study: An English Auction 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 [17, 18, 19] to perform this task.

William Vickrey [95] 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 [73, 74, 75] 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 outbid each other, thus effectively lowering the winning bid).

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

3.2.1 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 hasn't 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 specific 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.

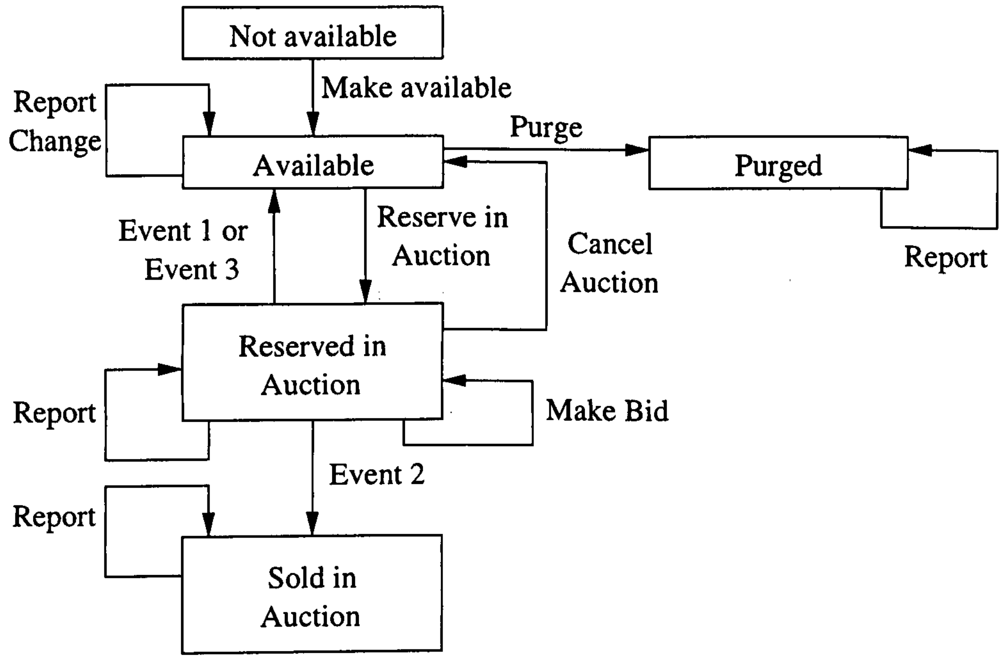

Figure 3.2: An English Auction Site - The life cycle graph of product's item

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 defined 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 figure 3.2: *Not Available*, *Available*, *Reserved in Auction*, *Sold in Auction* and *Purged*. The transitions in the graph can be seen in the figure. 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 defined in the methodology, Section 2.4.1, 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, defined in Table 3.1.

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

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

Table 3.1: English Auction Events

- 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.*
- 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.

3.2.2 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 2.4.2, 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 3.2. 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 verified.

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

3.2.3 Functional Level

Continuing the process defined 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 2.4.3, 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.

3.2.4 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.

Chapter 4

Conclusions and Future Work

This work 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 a virtual store and an English auction web site to demonstrate how the method works.

As a result of our virtual store 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. During this verification we have precisely identified both errors and their causes that would have been difficult to find out otherwise.

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, we call them model checking 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.

Appendix A

Overview of CAFE - Methodology for the Design of E-commerce Systems

CAFE [68] is a model developed to provide an incremental method to design e-commerce systems. It is oriented by trade goods, based on its life cycle. In this appendix we describe the methodology *CAFE*, which was the basis to develop this master thesis.

The start point of *CAFE* is the business-oriented model that will be provided. This model will not be discussed in details, but it must describes the nature of the products being negotiated and the type of negotiation. This description is normally textual and has the objective to distinguish the elements that are part of the business and the operations that are executed on them, which goes to be objects of the specification. *CAFE* is structured in four levels, as defined follow:

Conceptual: this level defines the entities of the e-commerce system. This entities are the basis to build the followed levels.

Application: this level models the life cycle of the products that are commercialized, identifying the types of operations (called actions) that are performed on the products and that change their states.

Functional: this level models the services provided by the system, detailing the user interaction with the system. In other words, the services implementation strategies are defined.

Execution: this level determines how the application will be implemented, considering the system's architecture, and define the requirements to be achieved.

This 0 structure aim to regard the phases of an e-commerce system conception. The first level, called conceptual, is very important because it defines the basis to build the other

Level	Components
Conceptual	Entities
Application	Product's item life cycle of the item Actions Agents
Functional	Services Products Product's items Functional requirements
Execution	System's architecture Components Protocols Tools

Table A.1: Elements of each level of *CAFE*'s methodology

ones. The application level focus on the life cycle of the product's item. The functional level detail this interactions. The last level, execution, complete the methodology, defining the architecture of the system. It's relevant to explain that this model is not rigid, so it's possible to define sub-phases to follow this process.

In the Table A.1 is presented some elements used to compose the e-commerce system specification.

In the following subsections we detail this levels.

A.1 Conceptual Level

In the same way that traditional models of commerce, the central object of the e-commerce are the products, goods or services, commercialized. For each commercialized product, it has one or more items, that are product instances. Each item is characterized by a state and the states are modified by actions executed by agents. Without loss of generality, virtual business is the environment where agents act to acquire products. The virtual business is implemented by an e-commerce server, that is responsible for managing the transactions involving products.

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 this 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.

A.2 Application Level

As mentioned, in the application level the business-oriented model starts to be detailed. Thus, on the basis of the business-oriented model, we extract the following information:

1. the types of products commercialized by the server;
2. the attributes that characterize the commercialized products;
3. the possible states that each item can assume;
4. the actions that can be executed and its effect on the state of items; and
5. the agents who execute actions.

All the entities and relations of the conceptual level are instantiated in the application level. It is important to stand out that, for each type of different product commercialized by the server, there is an instance of the application level.

An important definition, that is made in the application level, is the nature of the attributes of the commercialized items. Attributes can be characterized by the durability, the interval of time where a given instance of the attribute is not modified. The durability of an attribute is a function of its dynamic behavior, and the higher its frequency of update, the lower its durability will be. Thus, based on criterion durability, we can divide the attributes in two groups:

Static: attributes whose values do not change as consequence of actions. Its durability is theoretically infinite.

Dynamic: attributes whose values can be changed by actions, in accordance to the transition function. Notice that the dynamism of the attributes of one product can be varied, being able to define sub-groups of attributes depending on the implementation.

The attributes of an item define its state. Another form to represent the state of an e-commerce server is through the products and its states, where the state of a product is the amount of items in each possible state of the product. Both representations are equivalents, being also interchangeable.

Considering the nature of the attributes, we can distinguish three types of actions:

Innocuous actions: they do not affect the state of an item.

Temporary actions: they change the state of the item in temporary character, i.e, the item can assume its original state again, that is, its state before the application of the action, as effect of one or more actions.

Perennial actions: they change the state of the item in permanent character, being irreversible.

The life cycle of an item can be visualized through the graph of state transitions, where the vertices are states and the edges are actions that cause the state transition.

A.3 Functional Level

The functional level of the *CAFE* model describes the services to be offered and the components of the server. Before considering the method to describe these services, it is necessary to define the types of processing to be executed and to categorize the requirements in terms of data access and modification. After these definitions, we present the three parts that compose the functional level:

1. description of the services,

2. server architecture and

3. storage strategy.

A.3.1 Types of processing

The types of processing can be divided in accordance to the architecture of the components usually found in e-commerce servers: processor/memory, disk access and network.

Without loss of generality, we can distinguish ten types of processing that are normally executed by e-commerce servers. Obviously these categories are not exhausting and aim to demonstrate the criteria to be used to distinguish them. Below we present these categories listed by component.

Processor and Memory

Arithmetic: accomplishment of operations involving integers and floating-point numbers.

Processing of characters: manipulation of strings of characters.

Support to transactions: synchronization between the components and competing operations on the same products. These operations are necessary to keep the ACID properties of e-commerce servers.

Temporary storage: allocation and access to dynamic memory for data response and execution of other operations on them.

Disk

Reading of files: access and reading of data stored in files, such as static pictures and page skeletons generated by the e-commerce server.

Querying attribute-value: selection of a set of attributes of one or more products in accordance with a criterion of equality, similarity or coverage of values.

Writing of files: access and writing (insertions and modifications) of files, to possibly register some action of the customer.

Persistent storage: safe storage of data, in order to satisfy the transactional properties of isolation and durability.

Network

Access from the Internet: connectivity resources so that services can be accessed from the Internet.

Access Control: control of safe access to the e-commerce server, when the received and transmitted information are decoded and codified.

It must be standed out that the criterion of the component's architecture does not produce orthogonal categories. The access control, for example, even so has been fit as a network functionality, also can involve significant processing.

The distinction of these types of processing must also be based on limitations of current technologies and involved operations in the maintenance of transactional properties. The description of the types of processing demanded for each service serves to determine the technology to be used by the server. *Script* Languages, for example Tcl and Perl, are normally efficient in the manipulation of strings of characters, but they are inefficient in the accomplishment of arithmetical operations of floating-point, since all the variables are strings of characters and have to be converted when the operation is executed.

A.3.2 Data Categories

Functionally defining, the data are categorized under two criteria in order to subsidize its strategy of manipulation: volatileness, that quantifies the frequency of data update, and type of sharing, that describes how many services can access the data simultaneously and the protocol to access them.

Related to volatileness we can classify the data in four categories:

Static: the data do not change with the execution of actions on items. Descriptive information of products, as heading, author and synopsis of a book, are typically static.

Little volatile: data are modified eventually with the execution of actions. A typical example of little volatile data is the price of a product in a traditional virtual store, cause it is modified only when new remittances of products are received or the prices of the supplier change.

Very volatile: data are modified frequently as consequence of actions in the server. The amount of items in supply in a virtual store is an example of very volatile data, therefore it changes each time the product is bought.

Definitive: definitive data are those that, once modified by a given action, become static. The buyer of an item is a definitive data, therefore its content is modified only once.

Related to the type of sharing, the data can be classified in three categories:

Replicable: data you talked back can be copied and be used simultaneously to satisfy some requests. Static data are normally talked back.

Periodically consistent: data you consist periodically can be talked back, be accessed and be modified for more than one request simultaneously, but they are brought up to date in a global form eventually. Little volatile data as prices of products can be talked back and are consisted periodically when new lists of prices are put on use.

Mutually exclusive: data that are accessed for only one request at every moment. The inventory of a product is data that must be mutually exclusive, considering its importance for the trustworthiness of the business and its high degree of volatileness.

These two categorizations are not completely orthogonal, since we do not have all the 12 combinations of volatileness degree and type of sharing.

A.3.3 Services

The detailed specification of each service to be offered is the first component of the functional level of the *CAFE* model. The specification of a service consists of the following information:

Denomination: the identification of the service.

Description: brief description of the purpose and context of the service, which does not have to exceed a paragraph.

Parameters: which are the informations passed as parameters to the execution of a service.

Actions: which are the actions, as defined in the application level, that are executed by the occasion of a service.

Attributes: indication of the attributes that are modified by a service, including the origin of the new values that are designated.

Classification: the services are classified in accordance to the actions that compose them, more specifically, the most restrictive action indicates the category of the service. A service that executes perennial and innocuous actions is perennial, while a service that executes innocuous and temporary actions is temporary.

Data requirements: presentation of the requirements in terms of volatileness and form of sharing for each one of the attributes when each action is executed.

A.3.4 Functional Requirements

Since the services are described, is necessary to fit them in terms of the processing requirements that will be necessary for its execution, indicating the types of processing to be provided by the components of the e-commerce server.

The functional requirements characterize the demands to be taken by the components in accordance to the support given to the types of processing discussed in Section A.3.1, such as support to the processing of transactions, persistence of data and processing of strings of characters. The mapping properly said is the description of the services in terms of the components, justified on the basis of functional criteria and storage support.

It is important to stand out the commitments that must be considered when we select the components. For example, Database Management Systems(DBMS) provide persistence and atomicity in a safe form, however at high costs due to the complexity of the computation that has to be executed. Thus, when the support to the persistence of data, transactional properties and access control increases, also is higher the complexity of the computation to be done and, therefore, the latency to satisfy the requests of the customers.

A.3.5 Strategy of Storage

After the definition of the services and the features of the components that will execute them, we must determine in which component each product attribute will be stored and manipulated. The location of the attributes is function of attributes and components features. When an attribute can be stored in more than one component, the component that offers the best support, or the best relation cost-benefit must be selected.

The storage granularity is function of the accesses made by the services. We distinguish two types of access: horizontal and vertical. Horizontal accesses request some attributes of a given item, while vertical accesses select one or more attributes of several items. Thus, we can segment the data vertically or horizontally in accordance to the accesses done by the services. For example, when a customer makes a fetching in a virtual store, attributes of several items are selected, while the verification of an item for addition to the shopping cart segments the database in an horizontal form.

A.4 Execution Level

The execution level specifies in details the implementation of the e-commerce server. This specification is composed by five parts, described below. We must stand out that this structure aim to illustrate the detailing of inherent information to the execution level, once the format of the specification is dependent of the execution environment.

A.4.1 Server Architecture

The server architecture defines the nature of the software components being used and justifies their use, in terms of the functional requirements. For example, if one of the requirements is persistent storage, so the use of a database server is necessary. In the other hand, transactional support can be provided either by an application server or by the database server.

The definition of the architecture must consider not only functional aspects, but also financial aspects (cost) and existence of components. We must also define the communication protocols among the e-commerce servers components, expliciting the interaction mechanisms among the components.

A.4.2 Execution Environment

The execution environment describes the interconnection between the components and the customers interface. The definition of the execution environment must be coherent with the description of the services and functionalities (that compose the functional level). This description must be done in terms of paradigms of implementation, and system primitives. Examples of paradigms are client-server, remote procedure calls and message exchange. In terms of system primitives, we must enumerate resources such as TCP/IP support and ability to do *fork* and *rsh.*

A.4.3 Protocols

Once the types of communication and cooperation between the components are defined, it is necessary to specify the protocols to be used for this communication. Thus, for each service it must be indicated (in the case of standardized protocols) or be described the protocols used for communication between the components. Examples of standardized protocols are the protocol HTTP, standard ODBC, used to access the database manager systems(DBMS), and CGI interface, used for execution of dynamically instantiated tasks.

The definition of the protocols must specify the types, purpose and message format. The message type identifies the system primitive used (i.e., RPC, TCP/IP). The protocol can be briefly described by the time diagram of necessary messages for the achievement of the service. For each message the following information must be specified:

- purpose;
- type, that indicates the basic protocol to be used;
- sender and receiver;
- content; and
- format.

We can exemplify the specification of a protocol analyzing a service of distribution of banners (electronic announcements), used for paid advertisement in an e-commerce site. The announcements are generated by a specialized server who takes care of only one type of request, which receive a parameter that specifies the nature of the announcement. The attendance to the request consists of two messages, the first one specifying the features of the announcement to be generated and the second one containing the announcement. The messages are exchanged through a TCP/IP connection established between the requestor and the announcement server. The first message is textual and contains only the parameter to generate the announcement. This message is interpreted by the server, the announcement is generated and sent to the requestor through a binary message containing the picture corresponding to the announcement. All these information can be presented in a stream of messages diagram. In this diagram, there is a time line for each one of the participant entities of the protocol (requestor and server), that shows the time sequence of messages. Each arrow in the diagram represents a message that is exchanged and the notations indicate its purpose and its content.

A.4.4 Addressing

The e-commerce servers can use particular protocols, however, for use in the WWW, the several services must be accessed in a non-ambiguous and deterministic way. Thus, it is necessary to define an addressing standard that not only indicates the service being requested, but also the parameters that conduct the execution of the service. The addresses are normally URLs composed by three parts: identification of the server, specification of the service and its parameters. The identification of the server follows the standard of the URLs. The specification of the services follows the same convention of the static resources of the WWW, hierarchically organized. The parameters can be juxtaposed to the end of the URL in the case of a command GET or be explicitly listed by a command POST.

A.4.5 Tools

Once the services and the interaction between components are defined, it is necessary to define the tools that will be necessary to implement the server, in order to support the services and its implementation among the several components. Beyond its use in the implementation of the components, the cost of the employed tools must also be considered. Examples of tools are WWW servers, DBMSs and interface generators.

Appendix B

Overview of Formal Methods

Formal methods are a set of techniques of software engineering that use mathematical notation to describe the requirements of the system and detail forms to validate this specification and subsequent implementation.

The term formal methods refers to the use of mathematical modeling, calculation, and prediction in the specification, design, analysis, construction, and assurance of computer systems and software. The reason it is called "formal methods" rather than "mathematical modeling of software" is to highlight the nature of the mathematics involved.

The specification corresponds to one of the initial stages of software development process [40, 43] and its objective is to define, in a complete and consistent way, the functional requirements of the system. In general, the specification is written in natural language, being subjected to ambiguities, sensitivity to the context and different interpretations. Formal specification consists of the use of formal notations, based in mathematical techniques and formal logic, to specify systems. The use of formal notations and mathematical formalisms allows to reduce errors and ambiguities committed during this process, generating an accuracy and not ambiguous specification.

Unfortunately, as interest in formal methods increases, the number of misconceptions regarding formal methods continues to grow in tandem. While formal methods have been employed, to some extent, for over a quarter of a century, there are still very few people who understand exactly what formal methods are, and how they are applied in practice [9]. Many people completely misunderstand what constitutes a formal method, and how formal methods have been successfully employed in the development of complex systems.

In order to be familiar with formal methods, we looked for basic knowledge about it in related works. In [45], Hall presents seven old myths on the use of formal methods, aiming to show that these are not very real through practical examples. Seven widely held conceptions about formal methods are challenged. These beliefs are variants of the following:

1. formal methods can guarantee that software is perfect.
2. they work by proving that the programs are correct.
3. only highly critical systems benefit from their use.
4. they involve complex mathematics.
5. they increase the cost of development.
6. they are incomprehensible to clients.
7. nobody uses them for real projects.

The arguments are based on the author's experiences. They address the bounds of formal methods, identify the central role of specifications in the development process, and cover education and training.

This seven cited myths were analyzed by the author, who contributed with the following conclusions:

1. Formal methods are very helpful at finding errors early on and can nearly eliminate certain classes of error.
2. They work largely by making you think very hard about the system you propose to build.
3. They are useful for almost any application.
4. They are based on mathematical specifications, which are much easier to understand than programs.
5. They can decrease the cost of development.
6. They can help clients understand what they are buying.
7. They are being used successfully on practical projects in industry.

Myths that formal methods can guarantee perfect software and eliminate the need for testing (Myth 1 in Hall's paper) are not only ludicrous, but can have serious ramifications in system development if naive users of formal methods take them seriously. Claims that formal methods are all about proving programs correct (Myth 2 in Hall's paper) and are only useful in safety-critical systems (Myth 3), while untrue, are not quite so detrimental, and a number of successful applications in non safety-critical domains have helped to clarify these points (see [49] for examples).

The derivation of a number of simple formal specifications of quite complex problems, and the successful development of a number of formal methods projects under budget have served to dispel the myths that the application of formal methods requires highly trained mathematicians (Myth 4) and increases development costs (Myth 5). The successful participation of end-users and other non-specialists in system development with formal methods has ruled out the myth that formal methods are unacceptable to users (Myth 6), while the successful application of formal methods to a number of large-scale complex systems, many of which have received much media attention, should put an end to beliefs that formal methods are not used on real large-scale systems (Myth 7).

Regretfully, twelve years later, these and other misconceptions still abound. Formal methods are unfortunately the subject of extreme hyperbole or deep criticism in many of the "popular press" science journals. From the claims that the authors of such articles make, it is quite clear that they have little or no understanding of what formal methods are, nor how they have been applied in industry. In [9], seven more myths of formal methods are presented. In this work, the author analyzes these myths and tries to demystify them using examples. These myths are enumerated as follows:

1. Formal methods delay the development process.
2. Formal methods lack tools.
3. Formal methods replace traditional development methods.
4. Formal methods only apply to software.
5. Formal methods are unnecessary.
6. Formal methods are not supported.
7. Formal methods people always use formal methods.

Formal methods techniques provide many benefits in the process of systems development. The formal specification acts as a mechanism of fails prevention, through a precise specification and without ambiguity in the system's functional requirements. The initial stages of the system development (documentation, requirements specification, and design) are considered the most critical, whereas the incidence of fails is normally observed. It is a consensus that the fails introduced in the earliest stages of the system development's lifecycle are more difficult and expensive to be detected and removed.

B.1 Benefits of Using Formal Methods

The adoption of formal methods provides many benefits in the system development process. Some of them are described, as follows:

- Formal methods allow to properties and consequences of non-executable specification of the system to be analyzed through theorem-proving in the earlier stages of its lifecycle.
- Formal methods, through the use of mathematical formalisms, are capable to produce non-ambiguous and precise specification.
- Formal verification can be used to test initial specifications of the project, when normally is not possible to apply other validation technique.
- According to the automatic nature of formal verification, is possible to analyse changes in the specification in a reliable way.
- The use of automatic tools could reduce the amount of time demanded to develop the system.
- Early formal thought about the system.
- Easy derivation of test cases.

In spite of the high costs inherent in application of formal methods, the many benefits provided by its adoption can compensate the spent investments.

B.2 Issues and Choices in Formal Methods

Expertise in formal methods is not widespread, and can be costly to acquire. Furthermore, the resources available for any project are limited, so that effort expended on formal methods may reduce that available for other methods of analysis and assurance. For these reasons, formal methods need to be applied selectively. There are several dimensions in the use of formal methods that permit selective or partial application. According to [87], the most important ones are:

- The amount of formality can vary between occasional use of ideas and notation from discrete mathematics in a "pencil and paper" manner to "fully formal" treatments that are checked with a mechanical theorem prover.
- Formal methods can be applied to all, or only to selected, components of the system.

- Formal methods can be applied to selected properties of the system (e.g., absence of deadlock) rather than to its full functionality.
- Formal methods can be applied to all, or merely to some, of the stages of the development lifecycle. If the latter, we can choose whether to favor the earlier, or the later stages of the lifecycle.
- In all cases it is possible to include more or less detail and to choose the level of abstraction at which the formal treatment is conducted.

In [87], each of these is examined in more detail. In this work, he classified formal methods in levels, according to their formalism, as follows:

Level 0 : no formal method is applied. This corresponds to the common practice, where verification is a manual process of revision applied to documents written in natural language. The validation is done by tests.

Level 1 : formal methods use ideas and notation from discrete mathematics and logic, but within a loose framework, where mathematics, English, diagrams, and other notations are used together. Proofs are careful arguments that are evaluated by whether they persuade reviewers. This is the way most mathematics is done.

Level 2 : formal methods employ a fixed specification language for documenting requirements and designs. A specification language generally blends concepts from logic, discrete mathematics, and programming into a single notation. Often, the language is supported by tools that check specifications for certain types of errors, and that provide useful functions such as cross-referencing or typesetting. Analysis and proofs are performed by hand and recorded with pencil and paper, but make use of explicit axioms and proof rules that describe the semantics of the languages and methods used.

Level 3 : formal methods stress mechanized analysis. Their specification languages are generally closer to standard logic than those of type 2 formal methods, and are supported by tools that include proof checkers, theorem provers, or model checkers. The tools that support a Level 3 formal method are often referred to collectively as a verification system.

The advantage of Level 1 formal methods is the flexibility that is available: notations and techniques can be selected, or invented, to suit the particular problem at hand. These methods can be very effective when used by individuals or small teams possessed of skill and judgment, but the lack of standardized notation and methods can make communication and training difficult across larger groups.

Level 2 formal methods address the problems of communication and training by providing fixed specification notations([48, 92] and VDM [53] are well-known examples) and, usually, a methodology for using them. Individual Level 2 methods are well suited to some types of applications (e.g., data processing), and less well suited to others (e.g., concurrent systems); users must be careful not to stretch their chosen method beyond its limits.

In general, the Level 2 notations are optimized for descriptive, rather than analytic, purposes. If the goal is to use formal methods to calculate properties of a design for the purpose of analysis, then a Level 3 method equipped with appropriate tools will probably be more suitable. It generally requires considerable skill and experience to use Level 3 tools effectively, but they can provide a very high degree of assurance.

Depending on the kind of the application, the desire goals and the available resources, any alternative of these levels could be a good choice.

Another criterion that should be considered is the likely effectiveness of formal methods versus traditional methods for quality control and assurance. It is to be expected, and there is some evidence to support the expectation [61], that the intrinsically hard design problems tend to be the most prone to faults, and the most resistant to traditional means of assurance. These intrinsically hard problems generally involve complex interactions, such as the coordination of distributed, concurrent, or real-time computations, and redundancy management. It requires great skill to address these problems using formal methods, but the number and size of these problems may not be large. The greatest return on formal methods may be obtained when relatively few, very highly skilled people apply formal methods to the hardest and most critical problems.

In our work, we decide to apply model checking to design e-commerce systems, so we use the Level 3 of formal methods.

B.3 The Varieties of Formal Specifications

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.

According to [87], the formal specification languages could be categorized as:

- Model-oriented Specification
- Property-oriented Specification

- Specifications for Concurrent Systems

Each of this categories is next explained.

B.3.1 Model-oriented Specification

If specification or annotation of programs is the goal, then the formal notation employed should generally be close to, though more abstract than, that of programming, with operations changing values "stored" in an implicit system "state", with data structures described fairly concretely, and with control described in operational terms.

Formal notations with these characteristics are often described as model oriented, meaning that desired properties or behaviors are specified by giving a mathematical model that has those properties. For data structures, these models are often constructed from the notions of set theory using sets, functions, relations, and so on.

Some examples of model-oriented specification languages are VDM-SL [36], the language associated to VDM (Vienna Development Method), the specification language Z [3] and HOS [46] (Higher Order Software).

A disadvantage of model-oriented specifications is that they can be overly prescriptive: suggesting how something is to be implemented, rather than just the properties it is required to have. For example, even though the specification of the least function does not prescribe an algorithm, it is stated in terms of the pointer and array model, and so it would be fairly difficult to use this specification to establish correctness of an implementation that used linked lists instead.

B.3.2 Property-oriented Specification

In contrast to the model-oriented style of specification that is often preferred for program-level descriptions, specifications of early-lifecycle products such as requirements commonly use property-oriented notations. These notations use an axiomatic style to state properties and relationships that are required to hold of the component being described, without suggesting how it is to be achieved.

An advantage of property-oriented over model-oriented specifications is that it is possible merely to constrain certain relationships or values, without having to define them exactly. On the other hand, it is very easy to write conflicting constraints that cause the specification to become inconsistent; inconsistent specifications are unimplementable, and are very dangerous because they can be used to prove anything.

Property-oriented Specification languages are based on entities and attributes. The entities are key elements of software, which could be mapped in modules in a formal model. The attributes are specified by application of functions and relation to entities. They specify operations allowed to entities and relationships between them. The specification is determined in terms of axioms, which define the relationship between operations. The most known algebraic specification language is OBJ [40].

B.3.3 Specifications for Concurrent Systems

Concurrent and distributed systems can be specified in a variety of styles. One style takes some form of communication as primitive and has programming-like features for sending and receiving values. This style has a model-oriented flavor and is often referred to as process algebra. Another style takes shared variables as the primitive means of communication and often uses temporal logic to allow specification that a property should hold "henceforth" or "eventually" on some or all execution paths. This style has a property-oriented flavor. Methods associated with a kind of analysis known as model checking use one type of description (a kind of state machine) to specify the system concerned, and another (a kind of temporal logic) to specify the properties required of it.

Further distinctions concern whether concurrent activities are considered to occur simultaneously ("true" concurrency) or alternately ("interleaving" concurrency), and whether consideration of time is restricted to the order in which events happen, or whether duration is considered ("real-time" logics). The most popular specification languages for concurrent systems is CSP [50] and CCS [57] (Calculus of Concurrent Systems).

Our developed methodology, *Formal-CAFE* could be classified as property-oriented specification because it is based on the life cycle of the the items on sale and properties defined from the business rules.

Appendix C

Overview of Model Checking

In this section we present a brief background on model checking and describe some scenarios where this technique is successfully employed for developing correct and robust systems.

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.

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.

An error trace can also result from incorrect modeling of the system or from an incorrect specification (often called a false negative). The error trace can also be useful in identifying and fixing these two problems. A final possibility is that the verification task will fail to terminate normally, due to the size of the model, which is to large to fit into the computer memory. In this case, it may be necessary to redo the verification after changing some of the parameters of the model checker or by adjusting the model (e.g., by using additional abstractions).

C.1 Temporal Logic and Model Checking

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 [51]. Although a number of different temporal logics have been studied, most have an operator like Gf that is true in the present if f is always true in the future (i.e., if f is globally true). To assert that two events e_1 and e_2 never occur at the same time, one would write $G(\neg e_1 \vee \neg e_2)$. Temporal logics are often classified according to whether time is assumed to have a *linear* or a *branching* structure.

Several researches, including Burstall [82] and Pnueli [2], have proposed using temporal logic for reasoning about computer programs. However, Pnueli [2] was the first to use temporal logic for reasoning about concurrency. His approach involved proving properties of the program under consideration from a set of axioms that described the behavior of the individual statements in the program. The method was extended to sequential circuits by Bochmann [41] and Malachi and Owicki [97]. Since proofs were constructed by hand, the technique was often difficult to use in practice.

The introduction of temporal-logic model checking algorithms by Clarke and Emerson [20, 39] in the early 1980s allowed this type of reasoning to be automated. Because checking that a single model satisfies a formula is much easier than proving the validity of a formula for all models, it was possible to implement this technique very efficiently. The algorithm developed by Clarke and Emerson for the branching-time logic CTL was polynomial in both the size of the model determined by the program under consideration and in the length of its specification in temporal logic. They also showed how fairness [42] could be handled without changing the complexity of the algorithm. This was an important step in that the correctness of many concurrent programs depends on some type of fairness assumption; for example, absence of starvation in a mutual exclusion algorithm may depend on the assumption that each process makes progress infinitely often.

At roughly the same time Quielle and Sifakis [52] gave a model checking algorithm for a subset of *CTL*, but they did not analyze its complexity. Later, Clarke, Emerson, and Sistla [21] devised an improved algorithm that was linear in the product of the length of the formula and the size of the state transition graph. The algorithm was implemented in the EMC model checker, which was widely distributed and used to check a variety of network protocols and sequential circuits [6, 21, 31, 63, 64, 65, 66]. Early model checking systems were able to check state transition graphs with between 10^4 and 10^5 states at a rate of about 100 states per second for typical formulas. In spite of these limitations, model checking systems were used successfully to find previously unknown errors in several published circuit designs.

Sistla and Clarke [1, 4] analyzed the model checking problem for a variety of temporal logics and showed, in particular, that for linear temporal logic (LTL) the problem was *PSPACE*-complete. Pnueli and Lichtenstein [72] reanalyzed the complexity of checking linear-time formulas and discovered that although the complexity appears exponential in the length of the formula, it is linear in the size of the global state graph. Based on this observation, they argued that the high complexity of linear-time model checking might still be acceptable for short formulas. The same year, Fujita [62] implemented a tableau based verification system for LTL formulas and showed how it could be used for hardware verification.

CTL* is a very expressive logic that combines both branching-time and linear-time operators. The model checking problem for this logic was first considered in a paper by Clarke, Emerson, and Sistla [37], where it was shown to be *PSPACE*-complete, establishing that it is the same general complexity class as the model checking problem for LTL. This result can be sharpened to show that CTL* and LTL model checking are of the same algorithmic complexity (up to a constant factor) in both the size of the state graph and the size of the formula. Thus, for purposes of model checking, there is no practical complexity advantage to restricting oneself to a linear temporal logic [35].

Alternative techniques for verifying concurrent systems have been proposed by a number of other researches. Many of these approaches use automata for specifications as well as for implementations. The implementation is checked to see whether its behavior conforms to that of the specification. Because the same type of the model is used for both implementation and specification, an implementation at one level can also be used as a specification for the next level of refinement. The use of language containment is implicit in the work of Kurshan [88], which ultimately resulted in the development of a powerful verifier called COSPAN [60, 81, 98]. Vardi and Wolper [69] first proposed the use of ω-automata (automata over infinite words) for automated verification. They showed how the linear temporal logic model checking problem could be formulated in terms of language containment between ω-automata. Other notions of conformance between automata have also been considered, including observational equivalence [24, 83], and various refinement relations [24, 84].

C.2 Symbolic Algorithms

In the original implementation of the model checking algorithm, transition relations were represented explicitly by adjacency lists. For concurrent systems with small numbers of processes, the number of states was usually fairly small, and the approach was often quite practical. In systems with many concurrent parts however, the number of states in the global state transition graph was too large to handle. In 1987, McMillan [11, 59], then a graduate student at Carnegie Mellon University, realized that by using a symbolic representation for the state transition graphs, much larger system could be verified. The new symbolic representation was based on Bryant's *orderer binary decision diagrams* (OBBDS) [85]. OBBDs provide a canonical form for boolean formulas that is often substantially more compact than conjunctive or disjunctive normal form, and very efficient algorithms have been developed for manipulating them. Because the symbolic representation captures some of the regularity in the state space determined by circuits and protocols, it is possible to verify systems with an extremely large number of states - many orders of magnitude larger than could be handled by the explicit-state algorithms. By using the original CTL model checking algorithm [20] of Clarke and Emerson with the new representation for state transition graphs, it became possible to verify some examples that had more than 10^{20} states [11, 59]. Since then, various refinements of the OBDD-based techniques by other researches have pushed the state count up to more than 10^{120} [55, 56].

The implicit representation is quite natural for modeling sequential circuits and protocols. Each state is encoded by an assignment of boolean values to the set of state variables associated with the circuit or protocol. The transition relation can therefore be expressed as a boolean formula in terms of two sets of variables, one set encoding the old state and the other encoding the new. This formula is then represented by a binary decision diagram. The model checking algorithm is based on computing fix-points of predicate transformers that are obtained from the transition relation. The fix-points are sets of states that represent various temporal properties of the concurrent systems. In the new implementations, both the predicate transformers and the fix-points are represented with OBDDs. Thus, it is possible to avoid explicitly constructing the state graph of the concurrent system.

The model checking system that McMillan developed as part of his doctoral dissertation thesis is called SMV [59]. It is based on a language for describing hierarchical finite-state concurrent systems. Programs in the language can be annotated by specifications expressed in temporal logic. The model checker extracts a transition system represented as an OBDD from a program in the SMV language and uses an OBDD-based search algorithm to determine whether the system satisfies its specification. If the transition system does not satisfy some specification, the verifier will produce an execution trace that shows why the specification is false. The SMV system has been widely distributed, and a large number of examples have now been verified with it. These examples provide convincing evidence that SMV can be used to debug real industrial designs.

An impressive example that illustrates the power of symbolic model checking is the verification of the cache coherence protocol described in the IEEE Futurebus+ standard (IEEE Standard 896.1-1991). Although development of the Futurebus+ cache coherence protocol began in 1988, all previous attempts to validate the protocol were based entirely on informal techniques. In 1992, researchers at Carnegie Mellon [28, 38] construct a precise model of the protocol in SMV language and then used SMV to show that the resulting transition system satisfied a formal specification of cache coherence. They were able to find a number of errors that were not previously detected and potential errors in the design of the protocol. This appears to be the first time that an automatic verification tool has been used to find errors in an IEEE standard.

One of the best indications of the power of the symbolic verification methods comes from studying how the CPU time required for verification grows asymptotically with larger and larger instances of the circuit or protocol. In many of the examples that have been considered by a variety of groups, this growth rate is a small polynomial in the number of components of the circuit [30, 55, 56].

A number of other researches have independently discovered that OBDDs can be used to represent large state-transition systems. Coudert, Berthet, and Madre [70] have developed an algorithm for showing equivalence between two deterministic finite-state automata by performing a breadth first search of the state space of the product automata. They use OBDDs to represent the transition functions of the two automata in their algorithm. Similar algorithms have been developed by Pixley [12, 13]. In addition, several groups including Bose and Fisher [89], and Coudert, Madre, and Berthet [71] have experimented with model checking algorithms that use OBDDs.

In related work Bryant, Seger and Beatty [30, 86] have developed an algorithm based on symbolic simulation for model checking in a restricted linear time logic. Specifications consist of precondition-postcondition pairs expressed in the logic. The precondition is used to restrict inputs and initial states of the circuit; the postcondition gives the property that the user wishes to check. Formulas in the logic have the form

$p_0 \wedge Xp_1 \wedge X^2p_2 \wedge ... \wedge X^{n-1}p_{n-1} \wedge X^np_n.$

The syntax of the formulas is highly restricted compared to most other temporal logics used for specifying programs and circuits. By limiting the class of formulas that can be handled, it is possible to check certain properties very efficiently.

C.3 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. One can never be sure whether the unexplored trajectories may contain fatal bugs. A very attractive alternative to simulation and testing is the approach of *formal verification* which conducts an *exhaustive exploration* off all possible behaviors of the system.

Symbolic model checking is a formal verification approach 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 some temporal logic. Formally, the model is a labeled state-transition graph M. The labels correspond to the values of the variables in the program, while the transitions correspond to the passage of time in the model.

Model checkers have been successfully applied to the verification of several large complex systems such as an aircraft controller [16], a robotics controller [15], and a distributed heterogeneous real-time system [91]. The key to the efficiency of the algorithms is the use of *binary decision diagrams* 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 [14, 16]. We claim that symbolic model check will be an efficient technique to the formal verification of e-commerce systems.

C.3.1 Binary Decision Diagrams

Binary decision diagrams (BDDs) are a canonical representation for boolean formulas [10]. A BDD is obtained from a binary decision tree by merging identical subtrees and eliminating nodes with identical left and right siblings. The resulting structure is a directed acyclic graph rather than a tree which allows nodes and substructures to be shared.

The internal vertices are labeled with boolean variables. Leaves are labeled with 0 and 1. Canonicity is ensured placing a strict total order on the variables as one traverses a path from "root" to "leaf". The edges are labeled with 0 or 1. For every truth assignment there is a corresponding path in the BDD such that at vertex x, the edge labeled 1 is taken if the assignment sets x to 1; otherwise, the edge labeled 0 is taken.

If the path end in the "leaf" labeled 0 then the formula will not be satisfied, conversely, if it end in the "leaf" labeled 1 then the formula will be satisfied - the assignment made to

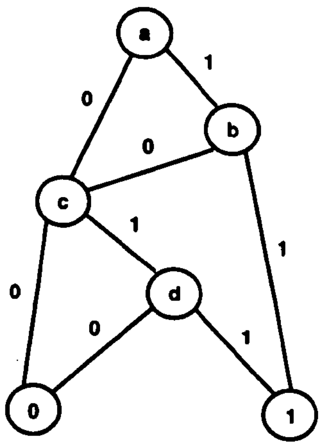

Figure C.1: BDD for $(a \wedge b) \vee (c \wedge d)$

each variable satisfies the formula. Figure C.1 illustrates the BDD for the boolean formula $(a \wedge b) \vee (c \wedge d)$.

C.4 Modeling Concurrent Systems

In order to model the system, a type of state transition graph called a *Kripke structure* is used. A Kripke structure consists of a set of states, a set of transitions between states, and a function that labels each state with a set of properties that are true in this state. Paths in a Kripke structure model computations of the system.

A state is a snapshot of the system that capture the values of the variables at a particular instant of time. An assignment of values to all the 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), $(\bar{a}, \bar{b}, c)$, and $(a, \bar{b}, \bar{c})$, respectively, where a means that the variable is true in the state and $\bar{a}$ means that the variable is false. Boolean formulas 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. One 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 $s = (a, \bar{b}, c)$ is represented by the formula $a \wedge \neg b \wedge c$. We say that $a \wedge \neg b \wedge c$ is the formula associated

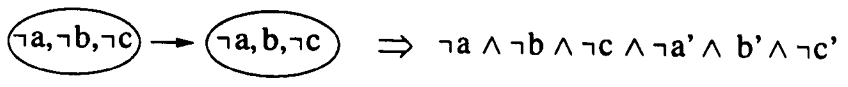

Figure C.2: Example of a transition and its symbolic representation.

with the state s. Because symbols are used to represent states, algorithms that use this method are called symbolic algorithms.

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 C.2) from the state $(\overline{a}, \overline{b}, \overline{c})$ to the state $(\overline{a}, b, \overline{c})$ is represented by the formula $\neg a \wedge \neg b \wedge \neg c \wedge \neg a' \wedge b' \wedge \neg c'$.

As boolean formulas can represent sets of states, they can also represent sets of transitions. Symbolic model checking takes advantage of this fact by grouping sets of transitions into a single formula which often significantly simplifies traversing the graph. Note that the transition relation of the model is a disjunction of all particular transitions in the graph. The clustering of transitions happens automatically when boolean formulas are implemented using BDDs. This occurs because bdds are canonicals: given a fixed variable ordering, a boolean formula is represented by a unique BDD [10]. Therefore, the order in which the transition relation is constructed does not affect the final result i.e., the canonical property guarantees that the same transitions will be clustered according to the formulas that represent them. This technique is one of the main reasons for the efficiency of symbolic algorithms.

In order to write specifications that describe properties of concurrent systems we need to define a set of *atomic proposition AP*. An atomic proposition is an expression that has the form v op d where $v \in V$ - the set of all variables in the system, $d \in D$ - the domain of interpretation, and op is any relational operator. Now, we can define formally a *Kripke structure M* over AP as a four tuple $M = (S, S_0, R, L)$ where:

1. S is a finite set of states.

2. $S_0 \subseteq S$ is the set of initial states.

3. $R \subseteq S \times S$ is a transition relation that must be total.

4. $L : 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.

```
procedure example;
var
    x := 0;
    y := 1;
begin
    while (true) x = y;
end
```

Figure C.3: A simple transition x = y

To illustrate the notions defined we consider the simple system in Figure C.3, where $V = \{x, y\}$, $D = \{0, 1\}$ and $S_0(x, y) \equiv (x = 0) \wedge (y = 1)$. The only possible transition is $x = y$ represented by the formula $R(x, y, x', y') \equiv (x' = y) \wedge (y' = y)$. The kripke structure $M = (S, S_0, R, L)$ extracted from these formula is:

- $S = \{(0,0), (0,1), (1,0), (1,1)\}$.
- $S_0 = \{(0,1)\}$.
- $R = \{[(0,0), (0,0)], [(0,1), (1,1)], [(1,0), (0,0)], [(1,1), (1,1)]\}$.
- $L((0,0)) = \{x = 0, y = 0\}$, $L((0,1)) = \{x = 0, y = 1\}$, $L((1,0)) = \{x = 1, y = 0\}$, and $L((1,1)) = \{x = 1, y = 1\}$.

The Figure C.4 graphically shows the kripke structure M. As one can note the only path which starts in the initial state is $(0,1)(1,1)(1,1)...(1,1)$. This is the only computation of the system.

C.5 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 C.5. 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

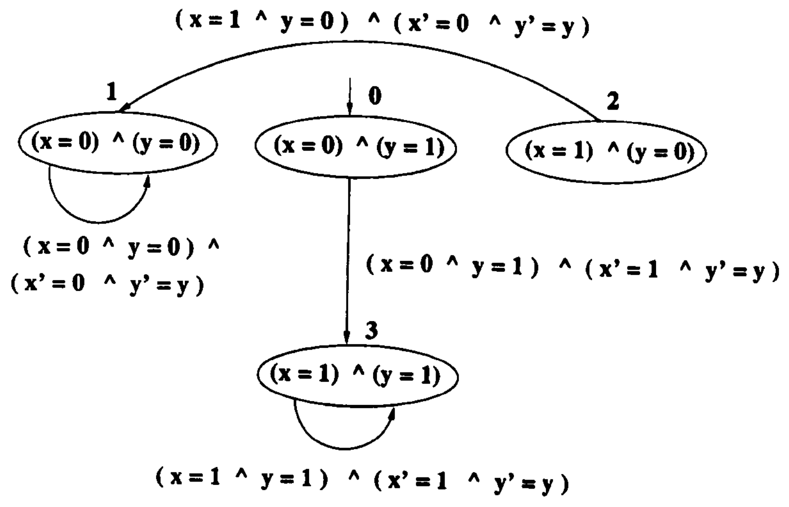

Figure C.4: The state transition graph

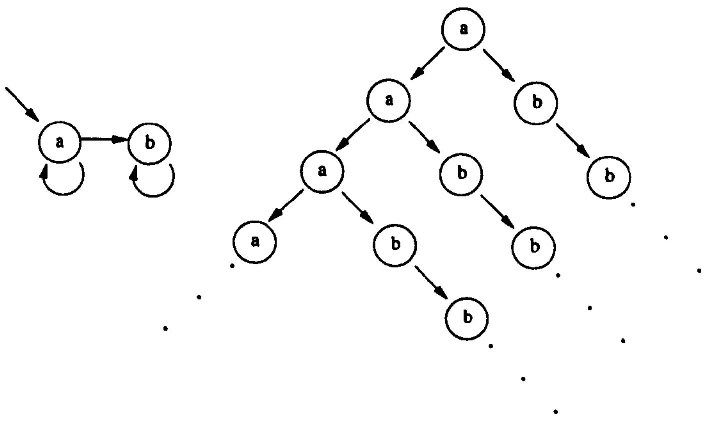

Figure C.5: State transition graph and corresponding computation tree.

$\wedge$, 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.

- **AG**(*req* $\rightarrow$ **AF** *ack*): It is always the case that if the signal *req* is high, then eventually *ack* will also be high.
- **EF**(*started* $\wedge \neg$*ready*): It is possible to get to a state where *started* holds but *ready* does not hold.

The four operators most widely used are illustrated in Figure C.6. Further details on

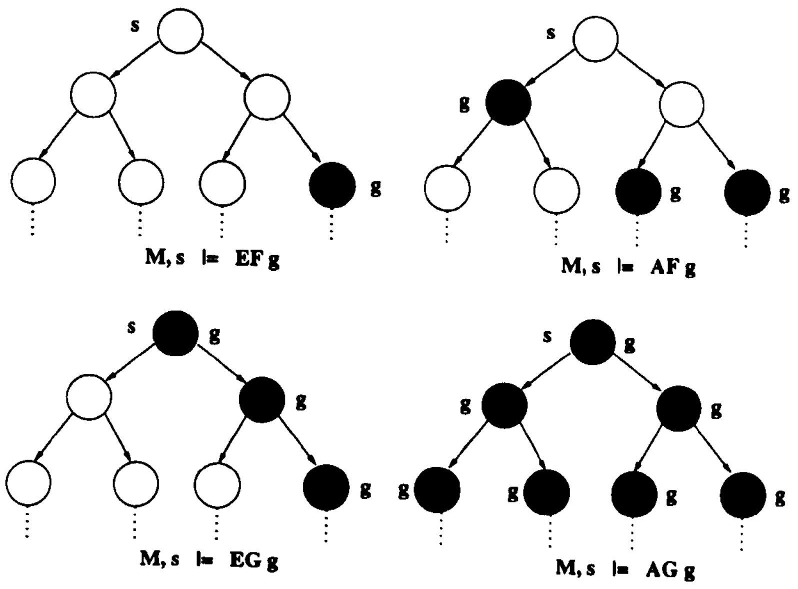

Figure C.6: Basic CTL operators

the semantics of the operators and on the expressiveness of the logic can be obtained in [91].

Appendix D

Overview of the *SMV* and *NuSMV* Systems

The SMV system [67] is a tool for checking finite state systems against specifications in the temporal logic CTL. The input language of SMV is designed to allow the description of finite state systems that range from completely synchronous to completely asynchronous, and from the detailed to the abstract. One can readily specify a system as a synchronous Mealy machine, or as an asynchronous network of abstract, nondeterministic processes. The language provides for modular hierarchical descriptions, and for the definition of reusable components. Since it is intended to describe finite state machines, the only basic data types in the language are finite scalar types. Static, structured data types can also be constructed. The logic CTL allows a rich class of temporal properties, including safety, liveness, fairness and deadlock freedom, to be specified in a concise syntax. SMV uses the OBDD-based symbolic model checking algorithm to efficiently determine whether specifications expressed in CTL are satisfied.

The primary purpose of the SMV input language is to provide a symbolic description of the transition relation of a finite Kripke structure. Any propositional formula can be used to describe this relation. This provides a great deal of flexibility, and at the same time a certain danger of inconsistency. For example, the presence of a logical contradiction can result in a deadlock - a state or states with no successor. This can make some specifications vacuously true, and makes the description unimplementable. While the model checking process can be used to check for deadlocks, it is best to avoid the problem when possible by using a restricted description style. The SMV system supports this by providing a parallel-assignment syntax. The semantics of assignment in SMV is similar to that of single assignment data flow languages. A program can be viewed as a system of simultaneous equations, whose solutions determine the next state. By checking programs for multiple assignments to the same variable, circular dependencies, and type errors, the compiler insures that a program using only the assignment mechanism is implementable. Consequently, this fragment of the language can be viewed as a hardware description language, or a programming language. The SMV system is by no means the last word on symbolic model checking techniques, nor is it intended to be a complete hardware description language. It is simply an experimental tool for exploring the possible applications of symbolic model checking to hardware verification.

D.1 An informal introduction

Before delving into the syntax and semantics of the language, let us first consider a few simple examples that illustrate the basic concepts. Consider the following short program in the language.

```
MODULE main

VAR
request : boolean;
state : {ready,busy};
ASSIGN
init(state) := ready;
next(state) := case
state = ready & request : busy;
1 : {ready,busy};
esac;

SPEC AG(request -> AF state = busy)
```

The input file describes both the model and the specification. The model is a Kripke structure, whose state is defined by a collection of state variables, which may be of Boolean or scalar type. The variable request is declared to be a Boolean in the above program, while the variable state is a scalar, which can take on the symbolic values ready or busy. The value of a scalar variable is encoded by the compiler using a collection of Boolean variables, so that the transition relation may be represented by an OBDD. This encoding is invisible to the user, however.

The transition relation of the Kripke structure, and its initial state (or states), are determined by a collection of parallel assignments (a system of simultaneous equations), which are introduced by the keyword ASSIGN. In the above program, the initial value of the variable state is set to ready. The next value of state is determined by the current state of the system by assigning it the value of the expression:

```
case
state = ready & request : busy;
1 : {ready,busy};
esac;
```

The value of a case expression is determined by the first expression on the right hand side of a (:) such that the condition on the left hand side is true. Thus, if state = ready & request is true, then the result of the expression is busy, otherwise, it is the set fready,busyg. When a set is assigned to a variable, the result is a non-deterministic choice among the values in the set. Thus, if the value of status is not ready, or request is false (in the current state), the value of state in the next state can be either ready or busy. Non-deterministic choices are useful for describing systems which are not yet fully implemented (i.e., where some design choices are left to the implementor), or abstract models of complex protocols, where the value of some state variables cannot be completely determined.

Notice that the variable request is not assigned in this program. This leaves the SMV system free to choose any value for this variable, giving it the characteristics of an unconstrained input to the system.

The specification of the system appears as a formula in CTL under the keyword SPEC. The SMV model checker verifies that all possible initial states satisfy the specification. In this case, the specification is that invariantly if request is true, then inevitably the value of state is busy.

The following program illustrates the definition of reusable modules and expressions. It is a model of a 3 bit binary counter circuit. Notice that the module name "main"has special meaning in SMV, in the same way that it does in the C programming language. The order of module definitions in the input file is inconsequential.

```
MODULE main
VAR
bit0 : counter.cell(1);
bit1 : counter.cell(bit0.carry.out);
bit2 : counter.cell(bit1.carry.out);
SPEC
```

```
AG AF bit2.carry.out

MODULE counter.cell(carry.in)
VAR
value : boolean;
ASSIGN
init(value) := 0;
next(value) := value + carry.in mod 2;
DEFINE
carry.out := value & carry.in;
```

In this example, we see that a variable can also be an instance of a user defined module. The module in this case is counter cell, which is instantiated three times, with the names bit0, bit1 and bit2. The counter cell module has one formal parameter carry in. In the instance bit0, this formal parameter is given the actual value 1. In the instance bit1, carryin is given the value of the expression bit0.carry out. This expression is evaluated in the context of the main module. However, an expression of the form a:b denotes component b of module a, just as if the module a were a data structure in a standard programming language. Hence, the carry in of module bit1 is the carry out of module bit0. The keyword DEFINE is used to assign the expression value & carry in to the symbol carry out. Definitions of this type are useful for describing Mealy machines. They are analogous to macro definitions, but notice that a symbol can be referenced before it is defined.

The effect of the DEFINE statement could have been obtained by declaring a variable and assigning its value, as follows:

```
VAR
carry.out : boolean;
ASSIGN
carry.out := value & carry.in;
```

Notice that in this case, the current value of the variable is assigned, rather than the next value. Defined symbols are sometimes preferable to variables, however, since they don't require introducing a new variable into the OBDD representation of the system. The weakness of defined symbols is that they cannot be given values non-deterministically. Another difference between defined symbols and variables is that while variables are statically typed, definitions are not. This may be an advantage or a disadvantage, depending on your point of view.

In a parallel-assignment language, the question arises: "What happens if a given variable is assigned twice in parallel?"More seriously: "What happens in the case of an absurdity, like a := a + 1; (as opposed to the sensible next(a) := a + 1;)?"In the case of SMV, the compiler detects both multiple assignments and circular dependencies, and treats these as semantic errors, even in the case where the corresponding system of equations has a unique solution. Another way of putting this is that there must be a total order in which the assignments can be executed which respects all of the data dependencies. The same logic applies to defined symbols. As a result, all legal SMV programs are realizable.

By default, all of the assignment statements in an SMV program are executed in parallel and simultaneously. It is possible, however, to define a collection of parallel processes, whose actions are interleaved arbitrarily in the execution sequence of the program. This is useful for describing communication protocols, asynchronous circuits, or other systems whose actions are not synchronized (including synchronous circuits with more than one clock). This technique is illustrated by the following program, which represents a ring of three inverting gates.

```
MODULE main
VAR
gate1 : process inverter(gate3.output);
gate2 : process inverter(gate1.output);
gate3 : process inverter(gate2.output);
SPEC
(AG AF gate1.out) & (AG AF !gate1.out)

MODULE inverter(input)
VAR
output : boolean;
ASSIGN
init(output) := 0;
next(output) := !input;
```

A process is an instance of a module which is introduced by the keyword process. The program executes a step by non-deterministically choosing a process, then executing all of the assignment statements in that process in parallel. It is implicit that if a given variable is not assigned by the process, then its value remains unchanged. Because the choice of the next process to execute is non-deterministic, this program models the ring of inverters independently of the speed of the gates. The specification of this program states that the output of gate1 oscillates (ie., that its value is infinitely often zero, and infinitely often 1). In fact, this specification is false, since the system is not forced to execute every process infinitely often, hence the output of a given gate may remain constant, regardless of changes of its input.

In order to force a given process to execute infinitely often, we can use a fairness constraint. A fairness constraint restricts the attention of the model checker to those execution paths along which a given CTL formula is true infinitely often. Each process has a special variable called running which is true if and only if that process is currently executing. By adding the declaration

```
FAIRNESS
running
```

to the module inverter, we can effectively force every instance of inverter to execute infinitely often, thus making the specification true.

One advantage of using interleaving processes to describe a system is that it allows a particularly efficient OBDD representation of the transition relation. We observe that the set of states reachable by one step of the program is the union of the sets of states reachable by each individual process. Hence, rather than constructing the transition relation of the entire system, we can use the transition relations of the individual processes separately and the combine the results [58, 59]. This can yield a substantial savings in space in representing the transition relation.

The alternative to using processes to model an asynchronous circuit would be to have all gates execute simultaneously, but allow each gate the non-deterministic choice of evaluating its output, or keeping the same output value. Such a model of the inverter ring would look like the following:

```
MODULE main
VAR
gate1 : inverter(gate3.output);
gate2 : inverter(gate2.output);
gate3 : inverter(gate1.output);
SPEC
(AG AF gate1.out) & (AG AF !gate1.out)

MODULE inverter(input)
VAR
```

```
output : boolean;
ASSIGN
init(output) := 0;
next(output) := !input union output;
```

The union operator allows us to express a nondeterministic choice between two expressions. Thus, the next output of each gate can be either its current output, or the negation of its current input - each gate can choose non-deterministically whether to delay or not. As a result, the number of possible transitions from a given state can be as high as 2n, where n is the number of gates. This sometimes (but not always) makes it more expensive to represent the transition relation. The relative advantages of interleaving and simultaneous models of asynchronous systems are discussed in [58].

As a second example of processes, the following program uses a variable semaphore to implement mutual exclusion between two asynchronous processes. Each process has four states: idle, entering, critical and exiting. The entering state indicates that the process wants to enter its critical region. If the variable semaphore is zero, it goes to the critical state, and sets semaphore to one. On exiting its critical region, the process sets semaphore to zero again.

```
MODULE main
VAR
semaphore : boolean;
proc1 : process user;
proc2 : process user;
ASSIGN
init(semaphore) := 0;
SPEC
AG !(proc1.state = critical & proc2.state = critical)

MODULE user
VAR
state : {idle,entering,critical,exiting};
ASSIGN
init(state) := idle;
next(state) := case
state = idle : {idle,entering};
state = entering & !semaphore : critical;
state = critical : {critical,exiting};
state = exiting : idle;
1 : state;
esac;

next(semaphore) := case
state = entering : 1;
state = exiting : 0;
1 : semaphore;
esac;

FAIRNESS
running
```

If any specification in the program is false, the SMV model checker attempts to produce a counterexample, proving that the specification is false. This is not always possible, since formulas preceded by existential path quantifiers cannot be proved false by a showing a single execution path. Similarly, sub-formulas preceded by universal path quantifier cannot be proved true by a showing a single execution path. In addition, some formulas require infinite execution paths as counterexamples. In this case, the model checker outputs a looping path up to and including the first repetition of a state.

In the case of the semaphore program, suppose that the specification were changed to

```
AG (proc1.state = entering -> AF proc1.state = critical)
```

In other words, we specify that if proc1 wants to enter its critical region, it eventually does. The output of the model checker in this case is shown in figure 3.1. The counterexample shows a path with proc1 going to the entering state, followed by a loop in which proc2 repeatedly enters its critical region and the returns to its idle state, with proc1 only executing only while proc2 is in its critical region. This path shows that the specification is false, since proc1 never enters its critical region. Note that in the printout of an execution sequence, only the values of variables that change are printed, to make it easier to follow the action in systems with a large number of variables.

Although the parallel assignment mechanism should be suitable to most purposes, it is possible in SMV to specify the transition relation directly as a propositional formula in terms of the current and next values of the state variables. Any current/next state pair is in the transition relation if and only if the value of the formula is one. Similarly, it is possible to give the set of initial states as a formula in terms of only the current state variables.

These two functions are accomplished by the TRANS and INIT statements respectively. As an example, here is a description of the three inverter ring using only TRANS and INIT:

```
MODULE inverter(input)
VAR
output : boolean;
INIT
output = 0
TRANS
next(output) = !input | next(output) = output
```

According to the TRANS declaration, for each inverter, the next value of the output is equal either to the negation of the input, or to the current value of the output. Thus, in effect, each gate can choose nondeterministically whether or not to delay. The use of TRANS and INIT is not recommended, since logical absurdities in these declarations can lead to unimplementable descriptions. For example, one could declare the logical constant 0 (false) to represent the transition relation, resulting in a system with no transitions at all. However, the flexibility of these mechanisms may be useful for those writing translators from other languages to SMV.

To summarize, the SMV language is designed to be flexible in terms of the styles of models it can describe. It is possible to fairly concisely describe synchronous or asynchronous systems, to describe detailed deterministic models or abstract nondeterministic models, and to exploit the modular structure of a system to make the description more concise. It is also possible to write logical absurdities if one desires to, and also sometimes if one does not desire to, using the TRANS and INIT declarations. By using only the parallel assignment mechanism, however, this problem can be avoided. The language is designed to exploit the capabilities of the symbolic model checking technique. As a result the available data types are all static and finite. No attempt has been made to support a particular model of communication between concurrent processes (e.g., synchronous or asynchronous message passing). In addition, there is no explicit support for some features of communicating process models such as sequential composition. Since the full generality of the symbolic model checking technique is available through the SMV language, it is possible that translators from various languages, process models, and intermediate formats could be created. In particular, existing silicon compilers could be used to translate high level languages with rich feature sets into a low level form (such as a Mealy machine) that could be readily translated into the SMV language.

```
MODULE main
VAR
gate1 : inverter(gate3.output);
gate2 : inverter(gate1.output);
gate3 : inverter(gate2.output);
SPEC
(AG AF gate1.out) & (AG AF !gate1.out)

specification is false

AG (proc1.state = entering -> AF proc1.s... is false:

.semaphore = 0
.proc1.state = idle
.proc2.state = idle

next state: [executing process .proc1]

next state:
.proc1.state = entering

AF proc1.state = critical is false:
[executing process .proc2]

next state:
[executing process .proc2]
.proc2.state = entering

next state:
[executing process .proc1]
.semaphore = 1
.proc2.state = critical

next state:
[executing process .proc2]

next state:
[executing process .proc2]
.proc2.state = exiting

next state:
.semaphore = 0
.proc2.state = idle
```

Figure D.1: Model checker output for semaphore example

D.2 The input language

This section describes the various constructs of the SMV input language, and their syntax.

D.2.1 Lexical conventions

An atom in the syntax described below may be any sequence of characters in the set A-Z,a-z,0-9,_,-, beginning with an alphabetic character. All characters in a name are significant, and case is significant. Whitespace characters are space, tab and newline. Any string starting with two dashes ("–") and ending with a newline is a comment. A number is any sequence of digits. Any other tokens recognized by the parser are enclosed in quotes in the syntax expressions below.

D.2.2 Expressions

Expressions are constructed from variables, constants, and a collection of operators, including Boolean connectives, integer arithmetic operators, and case expressions. The syntax of expressions is as follows.

```
expr ::

atom  ;; a symbolic constant
|number  ;; a numeric constant
|id  ;; a variable identifier
|"!" expr  ;; logical not
|expr1 "&" expr2 ;; logical and
|expr1 "|" expr2  ;; logical or
|expr1 "->" expr2  ;; logical implication
|expr1 "<->" expr2  ;; logical equivalence
|expr1 "=" expr2  ;; equality
|expr1 "<" expr2  ;; less than
|expr1 ">" expr2  ;; greater than
|expr1 "<=" expr2  ;; less that or equal
|expr1 ">=" expr2  ;; greater than or equal
|expr1 "+" expr2  ;; integer addition
|expr1 "-" expr2  ;; integer subtraction
|expr1 "*" expr2  ;; integer multiplication
|expr1 "/" expr2  ;; integer division
|expr1 "mod" expr2  ;; integer remainder
|"next" "(" id ")"  ;; next value
|set_expr  ;; a set expression
|case_expr ;; a case expression
```

An id, or identifier, is a symbol or expression which identifies an object, such as a variable or defined symbol. Since an id can be an atom, there is a possible ambiguity if a variable or defined symbol has the same name as a symbolic constant. Such an ambiguity is flagged by the compiler as an error. The expression next(x) refers to the value of identifier x in the next state (see Section D.2.3). The order of parsing precedence from high to low is

```
*,/
+,-
mod
=,!,?,!=,?=
!
&
|
->,<->
```

Operators of equal precedence associate to the left. Parentheses may be used to group expressions.

A case expression has the syntax

```
case_expr ::

"case"
expr_a1 ":" expr_b1 ";"
expr_a2 ":" expr_b2 ";"
...
"esac"
```

A case expression returns the value of the first expression on the right hand side, such that the corresponding condition on the left hand side is true. Thus, if expr_a1 is true, then the result is expr_b1. Otherwise, if expr_a2 is true, then the result is expr_b2, etc. If none of the expressions on the left hand side is true, the result of the case expression is the numeric value 1. It is an error for any expression on the left hand side to return a value other than the truth values 0 or 1.

A set expression has the syntax

```
set_expr ::
"{" val1 "," val2 "," ... "}"
| expr1 "in" expr2  ;; set inclusion predicate
| expr1 "union" expr2  ;; set union
```

A set can be defined by enumerating its elements inside curly braces. The elements of the set can be numbers or symbolic constants. The inclusion operator tests a value for membership in a set. The union operator takes the union of two sets. If either argument is a number or symbolic value instead of a set, it is coerced to a singleton set.

D.2.3 Declarations

The VAR declaration

A state of the model is an assignment of values to a set of state variables. These variables (and also instances of modules) are declared by the notation

```
decl :: "VAR"
atom1 ":" type1 ";"
atom2 ":" type2 ";"
...
```

The type associated with a variable declaration can be either Boolean, scalar, or a user defined module. A type specifier has the syntax

```
type :: boolean

| "{" val1 "," val2 "," ... "}"
| atom [ "(" expr1 "," expr2 "," ... ")" ]
| "process" atom [ "(" expr1 "," expr2 "," ... ")" ]

val :: atom || number
```

A variable of type boolean can take on the numerical values 0 and 1 (representing false and true, respectively). In the case of a list of values enclosed in set brackets (where atoms are taken to be symbolic constants), the variable is a scalar which can take any of these values. Finally, an atom optionally followed by a list of expressions in parentheses indicates an instance of module atom (see Section D.2.4). The keyword process causes the module to be instantiated as an asynchronous process (see Section D.2.6).

The ASSIGN declaration

An assignment declaration has the form

```
decl :: "ASSIGN"
dest1 ":=" expr1 ";"
dest2 ":=" expr2 ";"
...

dest :: atom
| "init" "(" atom ")"
| "next" "(" atom ")"
```

On the left hand side of the assignment, atom denotes the current value of a variable, init(atom) denotes its initial value, and next(atom) denotes its value in the next state. If the expression on the right hand side evaluates to an integer or symbolic constant, the assignment simply means that the left hand side is equal to the right hand side. On the other hand, if the expression evaluates to a set, then the assignment means that the left hand side is contained in that set. It is an error if the value of the expression is not contained in the range of the variable on the left hand side.

In order for a program to be implementable, there must be some order in which the assignments can be executed such that no variable is assigned after its value is referenced. This is not the case if there is a circular dependency among the assignments in any given process. Hence, such a condition is an error. In addition, it is an error for a variable to be assigned more than once simultaneously. To be precise, it is an error if:

1. the next or current value of a variable is assigned more than once in a given process, or

2. the initial value of a variable is assigned more than once in the program, or

3. the current value and the initial value of a variable are both assigned in the program, or

4. the current value and the next value of a variable are both assigned in the program, or

5. there is a circular dependency, or 6. the current value of a variable depends on the next value of a variable.

The TRANS declaration

The transition relation R of the model is a set of current state/next state pairs. Whether or not a given pair is in this set is determined by a Boolean valued expression, introduced by the TRANS keyword. The syntax of a TRANS declaration is

```
decl :: "TRANS" expr
```

It is an error for the expression to yield any value other than 0 or 1. If there is more than one TRANS declaration, the transition relation is the conjunction of all of TRANS declarations.

The INIT declaration

The set of initial states of the model is determined by a Boolean expression under the INIT keyword. The syntax of a INIT declaration is

```
decl :: "INIT" expr
```

It is an error for the expression to contain the next() operator, or to yield any value other than 0 or 1. If there is more than one INIT declaration, the initial set is the conjunction of all of the INIT declarations.

The SPEC declaration

The system specification is given as a formula in the temporal logic CTL, introduced by the keyword SPEC. The syntax of this declaration is

```
decl :: "SPEC" ctlform
```

A CTL formula has the syntax

```
ctlform ::
expr  ;; a Boolean expression
| "!" ctlform  ;; logical not
| ctlform1 "&" ctlform2  ;; logical and
| ctlform1 "--" ctlform2  ;; logical or
| ctlform1 "->" ctlform2  ;; logical implies
| ctlform1 "<->" ctlform2  ;; logical equivalence
| "E" pathform  ;; existential path quantifier
| "A" pathform  ;; universal path quantifier
```

The syntax of a path formula is

```
pathform ::
"X" ctlform  ;; next time
"F" ctlform  ;; eventually
"G" ctlform  ;; globally
ctlform1 "U" ctlform2  ;; until
```

The order of precedence of operators is (from high to low)

```
E,A,X,F,G,U
!
&
|
->,<->
```

Operators of equal precedence associate to the left. Parentheses may be used to group expressions. It is an error for an expression in a CTL formula to contain a next() operator or to return a value other than 0 or 1. If there is more than one SPEC declaration, the specification is the conjunction of all of the SPEC declarations.

The FAIR declaration

A fairness constraint is a CTL formula which is assumed to be true infinitely often in all fair execution paths. When evaluating specifications, the model checker considers path quantifiers to apply only to fair paths. Fairness constraints are declared using the following syntax:

```
decl :: "FAIR" ctlform
```

A path is considered fair if and only if all fairness constraints declared in this manner are true infinitely often.

The DEFINE declaration

In order to make descriptions more concise, a symbol can be associated with a commonly used expression. The syntax for this declaration is

```
decl :: "DEFINE"
atom1 ":=" expr1 ";"
atom2 ":=" expr2 ";"
...
```

When every an identifier referring to the symbol on the left hand side occurs in an expression, it is replaced by the value of the expression on the right hand side (not the expression itself). Forward references to defined symbols are allowed, but circular definitions are not allowed, and result in an error.

D.2.4 Modules

A module is an encapsulated collection of declarations. Once defined, a module can be reused as many times as necessary. Modules can also be parameterized, so that each instance of a module can refer to different data values. A module can contain instances of other modules, allowing a structural hierarchy to be built. The syntax of a module is as follows.

```
module ::
[ "OPAQUE" ]
"MODULE" atom [ "(" atom1 "," atom2 "," ... ")" ]
decl1
decl2
...
```

The optional keyword OPAQUE is explained in the section on identifiers. The atom immediately following the keyword MODULE is the name associated with the module. Module names are drawn from a separate name space from other names in the program, and hence may clash with names of variables and definitions. The optional list of atoms in parentheses are the formal parameters of the module. Whenever these parameters occur in expressions within the module, they are replaced by the actual parameters which are supplied when the module is instantiated.

A *instance* of a module is created using the VAR declaration (see Section D.2.3). This declaration supplies a name for the instance, and also a list of actual parameters, which are assigned to the formal parameters in the module definition. An actual parameter can be any legal expression. It is an error if the number of actual parameters is different from the number of formal parameters. The semantics of module instantiation is similar to call-by-reference. For example, consider the following program fragment:

```
...
VAR
a : boolean;
b : foo(a);
...
MODULE foo(x)
ASSIGN
x := 1;
```

The variable a is assigned the value 1. Now consider the following program:

```
...
DEFINE
a := 0;
VAR
b : bar(a);
...
MODULE bar(x)
DEFINE
a := 1;
y := x;
```

In this program, the value assigned to y is 0. Using a call-by-name (macro expansion) mechanism, the value of y would be 1, since a would be substituted as an expression for x.

Forward references to module names are allowed, but circular references are not, and result in an error.

D.2.5 Identifiers

An id, or identifier, is an expression which references an object. Objects are instances of modules, variables, and defined symbols. The syntax of an identifier is as follows.

```
id ::
atom
| id "." atom
```

An atom identifies the object of that name as defined in a VAR or DEFINE declaration. If a identifies an instance of a module, then the expression a:b identifies the component object named b of instance a. This is precisely analogous to accessing a component of a structured data type. Note that an actual parameter of module instance a can identify another module instance b, allowing a to access components of b, as in the following example:

```
...
VAR
a : foo(b);
b : bar(a);
...
MODULE foo(x)
DEFINE
c := x.p | x.q;

MODULE bar(x)
VAR
p : boolean;
q : boolean;
```

Here, the value of c is the logical or of p and q. If the keyword OPAQUE appears before a module definition, then the variables of an instance of that module are not externally accessible. Thus, the following program fragment is not legal:

```
...
VAR
```

```
a : foo();
DEFINE
b := a.x;
...
OPAQUE MODULE foo()
VAR
x : boolean;
...
```

D.2.6 Processes

Processes are used to model interleaving concurrency, with shared variables. A process is a module which is instantiated using the keyword process (see Section D.2.3). The program executes a step by nondeterministically choosing a process, then executing all of the assignment statements in that process in parallel, simultaneously. Each instance of a process has special variable Boolean associated with it called running. The value of this variable is 1 if and only if the process instance is currently selected for execution. The rule for determining whether a given variable is allowed to change value when a given process is executing is as follows: if the next value of a given variable is not assigned in the currently executing process, but is assigned in some other process, then the next value is the same as the current value.

D.2.7 Programs

The syntax of an SMV program is

```
program ::
module1
module2
...
```

There must be one module with the name main and no formal parameters. The module main is the one instantiated by the compiler.

To see a formal semantics assigned to SMV programs, I suggest the reading of Section 3.3 of McMillan's thesis [58].

D.3 The NuSMV System

In this work I used the NuSMV [17, 18, 19], a symbolic model checker jointly developed by Carnegie Mellon University (CMU) and Istituto per la Ricerca Scientifica e Tecnologica (IRST).

The *NuSMV* project aims at the development of an open architecture for model checking, which can be reliably used for the verification of industrial designs, as a core for custom verification tools, as a test-bed for formal verification techniques, and applied to other research areas.

NuSMV is available at *http://nusmv.irst.itc.it.*

The main features of *NuSMV* are the following:

- Functionalities:

  NuSMV allows for the representation of synchronous and asynchronous finite state systems, and the analysis of specifications expressed in Computation Tree Logic (CTL) and Linear Temporal Logic (LTL). Heuristics are available for achieving efficiency and partially controlling the state explosion. The interaction with the user can be carried on with a textual, as well as graphical, interface.

- Architecture:

  A software architecture has been defined. The different components and functionalities of *NuSMV* have been isolated and separated in modules. Interfaces between modules have been provided. This should allow to reduce the effort needed to modify and extend *NuSMV.*

- Quality of the implementation:

  NuSMV is written in ANSI C, is POSIX compliant, and has been debugged with Purify in order to detect memory leaks. Furthermore, the system code is thoroughly commented. *NuSMV* uses the state of the art BDD package developed at Colorado University. This makes it very robust, portable, efficient. Furthermore, its code should be easy to understand and modify by other people than the developers.

The input language of *NuSMV* is designed to allow the description of finite state machines (FSM) which range from completely synchronous to completely asynchronous, and from the detailed to the abstract.

One can readily specify a system as a synchronous Mealy machine, or as an asynchronous network of nondeterministic processes. The language provides for modular hierarchical descriptions, and for the definition of reusable components. Since it is intended to describe finite state machines, the only data types in the language are finite ones - booleans, scalars and fixed arrays. Static, data types can also be constructed.

Specifications can be expressed in CTL (Computation Tree Logic), or LTL (Linear Temporal Logic). These logics allow a rich class of temporal properties, including safety, liveness, fairness and deadlock freedom, to be specified in concise a syntax.

The primary purpose of the *NuSMV* input is to describe the transition relation of a Kripke structure. Any expression in the propositional calculus can be used to describe this relation. This provides a great deal of flexibility, and at the same time a certain danger of inconsistency. For example, the presence of a logical contradiction can result in a deadlock – a state or states with no successor. This can make some specifications vacuously true, and makes the description unimplementable. While the model checking process can be used to check for deadlocks, it is best to avoid the problem when possible by using a restricted description style. The *NuSMV* system supports this by providing a parallel-assignment syntax. The semantics of assignment in *NuSMV* is similar to that of single assignment data flow language. By checking programs for multiple parallel assignments to the same variable, circular assignments, and type errors, the interpreter insures that a program using only the assignment mechanism is implementable. Consequently, this fragment of the language can be viewed as a description language, or a programming language.

Bibliography

[1] A. P. Sistla. *Theoretical Issues in the Design of Distributed and Concurrent Systems*. PhD thesis, Harvard University, Cambridge, MA, 1983.

[2] A. Pnueli. The temporal logic of programs. *In Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science*, pages 46–57, 1977.

[3] J. Abrial. The specification language z : Basic library, 1980.

[4] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logics. *Journal of Assoc. Comput. Mach.*, 32(3):733–749, July 1985.

[5] G. S. Avrunin, U. A. Buy, J. C. Corbett, L. K. Dillon, and J. C. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Transactions on Software Engineering*, 17(11):1204–1222, Nov. 1991.

[6] B. Mishra and E.M. Clarke. Hierarchical verification of asynchronous circuits using temporal logic. *Theoretical Computer Science*, 38:269–291, 1985.

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

[8] J. P. Bowen. Formal methods in safety-critical standards. In *Proc. 1993 Software Engineering Standards Symposium (SESS'93), Brighton, UK*, pages 168–177. IEEE Computer Society Press, 30 – 3 1993.

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

[10] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[11] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model-checking: 10 20 states and beyond, 1992.

[12] C. Pixley, G. Beihl, and E. Pacas-Skewes. Automatic derivation of FSM specification to implementation encoding. In *Proceedings of The International Conference on Computer Design*, pages 245–249, Cambridge, MA, Oct. 1991.

[13] C. Pixley, S.-W. Jeong, and G.D. Hachtel. Exact calculation of synchronization sequences based on binary decision diagrams. In *29th ACM/IEEE Design Automation Conference*, pages 620–623, 1992.

[14] 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.

[15] 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.

[16] 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.

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

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

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

[20] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic, 1981.

[21] E. Clarke and E. Emerson. Sistla: Automatic verification of finite-state concurrent systems using temporal logic specification, 1986.

[22] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. In L. Claesen, editor, *International Symposium on Computer Hardware Description Languages an d their Applications*. North-Holland, April 1993.

[23] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[24] R. Cleaveland, 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, January 1993.

[25] D. Krishnamurty and J. Rolia. Predicting the performance of an e-commerce server: Those mean percen tiles. In *Proc. First Workshop on Internet Server Performance ACM SIGME TRICS*, July 1998.

[26] 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.

[27] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. *In 1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages pages 522–525, 1992.

[28] D.E. Long. *Model Checking, Abstraction, and Compositional Verification*. PhD thesis, Carnegie Mellon University, 1993.

[29] S. L. 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.

[30] D.L. Beatty, R.E. Bryant, and C.-J.H. Seger. Formal Hardware Verification by Symbolic Ternary Trajectory Evaluation. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*. IEEE, IEE Computer Society Press, June 1991.

[31] D.L. Dill and E.M. Clarke. Automatic verification of asynchronous circuits using temporal logic. *IEE Proceedings*, 133 Part E(5):276–282, Sept. 1986.

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

[33] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *21st International Conference on Software Engineering*, May 1999.

[34] M. B. Dwyer and L. 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, Dec. 1994.

[35] E.A. Emerson and C.-L. Lei. Modalities for model checking: Branching time strikes back. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, New York, Jan. 1985. ACM.

[36] R. Elmstrøm, P. G. Larsen, and P. B. Lassen. The IFAD VDM-SL toolbox: A practical approach to formal specifications. *ACM SIGPLAN Notices*, 29(9):77–80, 1994.

[37] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic. In *Proceedings of the tenth Annual ACM Symposium on Principles of Programming Languages*, 1983.

[38] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the Futurebus+ Cache Coherence Protocol. In D. Agnew, L. Claesen, and R. Camposano, editors, *The Eleventh International Symposium on Computer Hardware Description Languages and their Applications*, pages 5–20, Ottawa, Canada, Apr. 1993. IFIP WG10.2, CHDL'93, IEEE COMPSOC, Elsevier Science Publishers B.V., Amsterdam, Netherland.

[39] E. Emerson. Branching time temporal logics and the design of correct concurrent programs, 1981.

[40] R. E. Fairley. *Software Engineering Concepts*. McGraw-Hill, New York, New York, 1985.

[41] G. V. Bochmann. Hardware specification with temporal logic: An example. *IEEE Transactions on Computers*, C-31(3):46–57, 1982.

[42] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, NV, 163-173*, 1980.

[43] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Englewood Cliffs, 1991.

[44] S. Gurgens, J. Lopez, and R. Peralta. Efficient detection of failure modes in electronic commerce protocols. In *DEXA Workshop*, pages 850–857, 1999.

[45] J. A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990.

[46] M. Hamilton and S. Zeldin. Higher order software - a methodology for defining software. *Software Engineering*, 2(1):9–32, 1976.

[47] Z. Har'El and R. Kurshan. Software for analytical development of communications protocols. *AT&T Technical Journal*, 69(1):45–59, Jan. 1990.

[48] I. J. Hayes, editor. *Specification Case Studies*. Prentice Hall International Series in Computer Science, 2nd edition, 1993.

[49] M. G. Hinchey and J. P. Bowen. Applications of formal methods. *Prentice Hall, first edition*, 1995.

[50] C. Hoare. Communicating sequential processes: Prentice-hall international, 1985.

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

[52] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in cesar. *In Proceedings of the fifth International Symposium on Programming*, 1981.

[53] C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990. ISBN 0-13-880733-7.

[54] W. M. Jr., C. D. Murta, S. V. A. Campos, and D. O. G. Neto. *Sistemas de Comercio Eletronico, Projeto e Desenvolvimento*. Campus, 2002.

[55] J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P.B. Denyer, editors, *International Conference on Very Large Scale Integration*, pages 49–58, Edinburgh, Scotland, Aug. 1991. IFIP Transactions, North-Holland.

[56] J.R. Burch, E.M. Clarke, D.E. Long, K.L. MacMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, Apr. 1994.

[57] P. Kanellakis, S. Smolka, E. Finite, P. Three, and o Equivalence. Information and computation, 1990.

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

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

[60] R. P. Kurshan. *Computer-aided Verification of Coordinating Processes - The Automata-Theoretic Approach*. Princeton Univ. Press, 1994.

[61] R. R. Lutz. Analyzing software requirements errors in safety-critical, embedded systems. Technical Report TR92-27, Iowa State University, Aug. 27, 1992.

[62] M. Fujita, H. Tanaka, and T. Moto-oka. Logic design assistance with temporal logic. In C.J. Koomen and T. Moto-oka, editors, *Proceedings of the Seventh International Symposium on Computer Hardware Description Languages and Their Applications*, pages 129–138, Amsterdam, 1983. IFIP, North-Holland.

[63] M.C. Browne and E.M. Clarke. SML: A high level language for the design and verification of finite state machines. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 269–292, Amsterdam, 1987. North-Holland.

[64] M.C. Browne, E.M. Clarke, and D.L. Dill. Checking the correctness of sequential circuits. In *Proceedings of the IEEE's International Conference on Computer Design*, pages 445–448, Port Chester, New York, 1985.

[65] M.C. Browne, E.M. Clarke, and D.L. Dill. Automatic circuit verification using temporal logic: Two new examples. In G.J. Milne and P.A. Subrahmanyam, editors, *Formal Aspects in VLSI Design*, pages 113–124, Amsterdam, 1986. North-Holland.

[66] M.C. Browne, E.M. Clarke, D.L. Dill, and B. Mishra. Automatic Verification of Sequential Circuits Using Temporal Logic. *IEEE Transactions on Computers*, C-35(12):1035–1044, 1986.

[67] K. McMillan. The smv system draft, 1992.

[68] W. Meira Jr., C. Murta, S. Campos, and D. Guedes. *Comércio Eletrônico: Projeto e Desenvolvimento de Sistemas.* Edições SBC–Campus. Campus, Rio de Janeiro, RJ, 2002.

[69] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the Annual Symposium on Logic in Computer Science*, pages 332–344. IEEE Computer Society Press, D.C., June 1986.

[70] O. Coudert, C. Berthet, and J.C. Madre. Verification of synchronous sequential machines using symbolic execution. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373, Grenoble, France, June 1989. Springer-Verlag.

[71] O. Coudert, C. Berthet, and J.C. Madre. Verifying temporal properties of sequential machines without building their state diagrams. In R. P. Kurshan and E. M. Clarke, editors, *Proceedings of the 1990 Workshop on Computer-Aided Verification*, June 1990.

[72] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 97–107, New York, Jan. 1985. ACM.

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

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

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

[76] G. Paixão, W. M. Jr., V. Almeida, D. Menascé, and A. Pereira. Design and implementation of a tool for measuring the performance of complex e-commerce sites. pages 309–323, March 2000. Performance Tools 2000 - Motorola University, Illinois - USA.

[77] A. Pereira, B. Gontijo, T. Cançado, and W. Meira Jr. Replicação de dados em servidores paralelos de comércio eletrônicos. In *Anais do I Workshop em Sistemas Computacionais de Alto Desempenho*, pages 45–50, São Pedro - SP, Outubro 2000.

[78] 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, Oct. 2002. Springer-Verlag.

[79] A. Pereira, M. Song, G. Gorgulho, W. Meira Jr., and S. Campos. Uma metodologia para verificação de modelos de sistemas de comércio eletrônico. In *Proceedings of the 5th WORKSHOP ON FORMAL METHODS (WMF'2002)*, Lecture Notes in Computer Science, Gramado, RS, Brasil, Oct. 2002.

[80] 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.

[81] R. H. Hardin, Z. Har'El, and R. P. Kurshan. COSPAN. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 423–427, New Brunswick, NJ, USA, July/Aug. 1996. Springer Verlag.

[82] R. M. Burstall. Program proving as hand simulation with a little induction. *In IFIP Information Processing '74*, pages 308–312, 1974.

[83] R. Milner. An algebraic definition of simulation between programs. In *In Proceedings of the Second Internation Joint Conference on Artificial Intelligence*, Sept. 1971.

[84] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1980.

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

[86] R.E. Bryant and C.-J.H. Seger. Formal verification of digital circuits using symbolic ternary system models. In E.M. Clarke and R.P. Kurshan, editors, *Proceedings of the Workshop on Computer-Aided Verification (CAV90)*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, New York, 1991. American Mathematical Society, Springer-Verlag.

[87] J. Rushby. Formal methods and their role in the certification of critical systems, 1995.

[88] S. Aggarwal, R.P. Kurshan, and K.K. Sabnani. A Calculus for Protocol Specification and Validation. In *Protocol Specification, Testing and Verification III*, pages 19–34, Amsterdam, 1983. North-Holland.

[89] S. Bose and A.L. Fisher. Automatic Verification of Synchronous Circuits Using Symbolic Simulation and Temporal Logic. In L.J.M. Claesen, editor, *Proceedings of the IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, Nov. 1989.

[90] S. Campos, E. Clarke, W. Marrero and M. Minea. Verus: a tool for quantitative analysis of finite-state real-time systems. *Workshop on Languages, Compilers and Tools for Real-Time Systems*, 1995.

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

[92] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 2nd edition, 1992.

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

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

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

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

[97] Y. Malachi and S. S. Owicki. Temporal specifications of self-timed systems. *In H. T. Kung, B. Sproull, and G. Steele, editors, VLSI Systems and Computations.*, 1981.

[98] Z. Har'El and R.P. Kurshan. Software for analytical development of communication protocols. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, Jan. 1990.

Appendix A

Source Code of Virtual Store Case Study

A.1 Application Level

```

-- Virtual Store - Application Level
-- Adriano Machado
-- Version: 1.3

MODULE main

VAR
        buyer: process buyer_agent(1);  -- buyer agent process
        seller : process  seller_agent (1);   -- seller agent process
        item: process item(1,buyer.action, seller .action);  -- item process

SPEC EF (item.state = Not_Available)
SPEC EF (item.state = Available)
SPEC EF (item.state = Reserved)
SPEC EF (item.state = Sold)
SPEC EF (item.state = Purged)

SPEC EF (buyer.action = Reserve)
SPEC EF (buyer.action = Report)
SPEC EF (buyer.action = Cancel_Reserve)
SPEC EF (buyer.action = Buy)
SPEC EF (buyer.action = none)

SPEC EF (seller.action = Make_Available)
SPEC EF (seller.action = Purge)
SPEC EF (seller.action = Change)
SPEC EF (seller.action = none)

-- Module that represents the item

MODULE item(item_id, action1, action2)

VAR
    state : {Not_Available, Available, Reserved, Sold, Purged};

ASSIGN

init(state) := Not_Available;

next(state) := case
        state = Not_Available & action2=Make_Available: Available;
        state = Available & action1=Report : Available ;
        state = Available & action1=Reserve : Reserved;
        state = Available & action2=Change : Available;
        state = Available & action2=Purge : Purged;
        state = Reserved & action1=Cancel_Reserve : Available;
        state = Reserved & action1=Buy : Sold;
        1: state;
  esac;

FAIRNESS running

-- Module that represents the seller agent

MODULE seller_agent(seller_id)

VAR
        action: {Make_Available, Change, Purge, none};
```

```

ASSIGN

init (action) := none;

next(action) := case
        action = Make_Available : {Make_Available, Change, Purge,none};
        action = Change : {Make_Available, Change, Purge,none};
        action = Purge : {Make_Available, Change, Purge,none};
        action = none : {Make_Available, Change, Purge, none};
        1: action;
esac;

FAIRNESS running

-- Module that represents the buyer agent

MODULE buyer_agent(id)

VAR
        action: {Report, Reserve, Cancel_Reserve, Buy, none};

ASSIGN

init (action) := none;

next(action) := case
        action = Report : {Report,Reserve,Cancel_Reserve,Buy,none};
        action = Reserve : {Report,Reserve,Cancel_Reserve,Buy,none};
        action = Cancel_Reserve : {Report,Reserve,Cancel_Reserve,Buy,none};
        action = Buy : {Report,Reserve,Cancel_Reserve,Buy,none};
        action = none : {Report, Reserve, Cancel_Reserve, Buy, none};
        1: action;
esac;

FAIRNESS running
```

A.2 Functional Level

```

-- Virtual Store - Functional Level
-- Adriano Machado
-- Version: 2.7

MODULE main

VAR
        buyer1: process buyer_agent(1);
        seller1 : process seller_agent (1);
        pr1: process product(1,buyer1.action, seller1 .action);
        it1 : process item(1,pr1.item_id,buyer1.action, seller1 .action);
        it2 : process item(2,pr1.item_id,buyer1.action, seller1 .action);

SPEC EF(pr1.bits_array = 0)
SPEC EF(pr1.bits_array = 1)
SPEC EF(pr1.bits_array = 2)
SPEC AF(pr1.bits_array = 2)
SPEC EF(pr1.bits_array = 3)
SPEC EF(pr1.item_id = 0)
SPEC EF(pr1.item_id = 1)
SPEC EF(pr1.item_id = 2)
SPEC AG(pr1.item_id = 0 | pr1.item_id = 1 | pr1.item_id = 2)
SPEC AG(pr1.item_id = 0 | pr1.item_id = 1)
SPEC AG(pr1.bits_array = 2 -> AX pr1.item_id = 2)
SPEC EF(it2.state = Available & buyer1.action=Reserve)
SPEC EF(it2.state = Available & buyer1.action=Reserve & pr1.item_id = 2)
SPEC EF(it1.state = Available & buyer1.action=Reserve & pr1.item_id = 1)

-- First Level Properties
SPEC EF (it1.state = Not_Available)
SPEC EF (it1.state = Available)
SPEC EF (it1.state = Reserved)
SPEC EF (it1.state = Sold)
SPEC EF (it1.state = Purged)

SPEC EF (it2.state = Not_Available)
SPEC EF (it2.state = Available)
SPEC EF (it2.state = Reserved)
SPEC EF (it2.state = Sold)
SPEC EF (it2.state = Purged)

SPEC EF (buyer1.action = Reserve)
SPEC EF (buyer1.action = Report)
SPEC EF (buyer1.action = Cancel_Reserve)
SPEC EF (buyer1.action = Buy)
SPEC EF (buyer1.action = none)

SPEC EF (seller1.action = Make_Available)
SPEC EF (seller1.action = Purge)
SPEC EF (seller1.action = Change)
SPEC EF (seller1.action = none)

-- Consistency Examples
SPEC AG !(pr1.inventory > 2)
SPEC AG !(pr1.inventory < 0)
SPEC AG !(pr1.inventory = 0 & pr1.productIsAvailable = 1)
SPEC AG !(pr1.inventory > 0 & pr1.productIsAvailable = 0)
SPEC EF !(pr1.bits_array = 2 & pr1.item_id=1)
SPEC EF !(pr1.bits_array = 1 & pr1.item_id=2)
SPEC EF !(pr1.bits_array = 0 & pr1.item_id != 0)
SPEC EF !(pr1.bits_array = 3 & pr1.item_id = 0)

-- Isolation Example
SPEC AG ((buyer1.action=Reserve & seller1.action=Make_Available) -> AX ((pr1.
    inventory > 0 & pr1.productIsAvailable = 1) | (pr1.inventory = 0 & pr1.
    productIsAvailable = 0)))

-- Atomicity Example
SPEC ((it1.state=Available & it2.state=Reserved & buyer1.action=Reserve & seller1.
    action=none) -> AX (it1.state=Reserved & pr1.inventory = 0))

MODULE product(id_produto,op1,op2)

```

```
VAR
        item_id  : 0..2;
        inventory : 0..2;
        bits_array : 0..3; -- Each bit corresponds to an item
        productIsAvailable: boolean;

ASSIGN

init (productIsAvailable) := 0;

init ( bits_array ) := 0;
init (item_id) := 0;
init (inventory) := 0;

next(inventory) := case
        op2 = Make_Available & inventory = 1 : inventory + 1;
        op2 = Make_Available & inventory = 0 : inventory + 2;
        op1 = Reserve & inventory > 0: inventory - 1;
        op1 = Cancel_Reserve & inventory < 2: inventory + 1;
        1: inventory;
esac;

next(bits_array) := case
        bits_array = 0 & op2 = Make_Available & inventory = 0: bits_array + 3;
        bits_array = 0 & op1 = Cancel_Reserve : bits_array + item_id;
        bits_array = 1 & op1 = Cancel_Reserve & item_id =2: bits_array + item_id;
        bits_array = 2 & op1 = Cancel_Reserve & item_id =1: bits_array + item_id;
        bits_array = 3 & op1 = Reserve : bits_array - item_id;
        bits_array = 2 & op1 = Reserve & item_id =2: bits_array - item_id;
        bits_array = 1 & op1 = Reserve & item_id =1: bits_array - item_id;
        1: bits_array;
esac;

next(item_id) := case
        bits_array = 0 : item_id = 0; -- none of the items will be the target of the action
        bits_array = 1 : item_id = 1; -- action will be executed on item 1
        bits_array = 2 : item_id = 2; -- action will be executed on item 2
```

```
            bits_array = 3 : item_id = {1,2}; -- action will be executed on item 1 or 2
            1: item_id;
esac;

next(productIsAvailable) := case
          productIsAvailable = 0 & op2=Make_Available & inventory < 2 : 1;
          productIsAvailable = 0 & inventory > 0 : 1;
          productIsAvailable = 1 & inventory = 0 : 0;
          productIsAvailable = 0 & op1=Reserve : 0;
          productIsAvailable = 0 & op1=Cancel_Reserve & inventory < 2 : 1;
          productIsAvailable = 0 & op1=Buy : 0;
          productIsAvailable = 0 & op2=Change : 0;
          productIsAvailable = 1 & op1=Report : 1 ;
          productIsAvailable = 1 & op1=Reserve & inventory = 1 : 0;
          productIsAvailable = 1 & op1=Cancel_Reserve : 1;
          productIsAvailable = 1 & op1=Buy : 1;
          productIsAvailable = 1 & op2=Make_Available : 1;
          productIsAvailable = 1 & op2=Change : 1;
          productIsAvailable = 0 & inventory = 0 & op1=none: 0;
          productIsAvailable = 0 & inventory = 0 & op2=none: 0;
          1: productIsAvailable;
esac;

FAIRNESS running

-- Module that represents the item

MODULE item(id,iditem,op1,op2)

VAR
          state : {Not_Available, Available, Reserved, Sold, Purged};
ASSIGN

 init(state) := Not_Available;

next(state) := case
          state = Not_Available & op2=Make_Available: Available;
          state = Available & op1=Report : Available;
```

```
        state = Available & op1=Reserve & id = iditem: Reserved;
        state = Available & op1=Cancel_Reserve & id = iditem: Available;
        state = Available & op1=Buy : Available;
        state = Available & op2=Make_Available : Available;
        state = Available & op2=Change : Available;
        state = Available & op2=Purge : Purged;
        state = Reserved & op1=Report : Reserved ;
        state = Reserved & op1=Reserve & id = iditem: Reserved;
        state = Reserved & op1=Cancel_Reserve & id = iditem: Available;
        state = Reserved & op1=Buy & id = iditem: Sold;
        state = Reserved & op2=Make_Available : Reserved;
        state = Reserved & op2=Change : Reserved;
        state = Sold & op1=Report : Sold;
        state = Sold & op1=Reserve : Sold;
        state = Sold & op1=Cancel_Reserve : Sold;
        state = Sold & op1=Buy : Sold;
        state = Sold & op2=Make_Available : Sold;
        state = Sold & op2=Change : Sold;
        1: state;
esac;

FAIRNESS running

-- Module that represents the seller agent

MODULE seller_agent(id)

VAR
        action: {Make_Available, Change, Purge, none};

ASSIGN

init(action) := Make_Available;

next(action) := case
        action = Make_Available : {Change, Purge, none};
        action = Change : {Change, Purge, none};
        action = Purge : {Change, none};
```

```
            action = none: {Change, Purge};
            1: action;
esac;

FAIRNESS running

-- Module that represents the buyer agent

MODULE buyer_agent(id)

VAR
            action: {Report, Reserve, Cancel_Reserve, Buy, none};

ASSIGN

init(action) := Report;

next(action) := case
            action = Report : {Report,Reserve,Buy, none};
            action = Reserve : {Report,Cancel_Reserve,Buy, none};
            action = Cancel_Reserve : {Report, Reserve, none};
            action = Buy : {Report, none};
            action = none : {Report, Reserve, Cancel_Reserve, Buy, none};
            1: action;
esac;

FAIRNESS running
```

A.3 Execution Level

```

-- Virtual Store - Architectural Level
-- Adriano Machado
-- Version: 7.6

MODULE main

VAR
```

```
    buyer1_www_socket: boolean;
    buyer2_www_socket: boolean;
    seller1_www_socket: boolean;
    www_shop_socket_1: boolean;
    www_shop_socket_2: boolean;
    www_shop_socket_3: boolean;
    shop_www_socket_1: boolean;
    shop_www_socket_2: boolean;
    shop_www_socket_3: boolean;
    www_buyer1_socket: boolean;
    www_buyer2_socket: boolean;
    www_seller1_socket: boolean;

VAR
        buyer1: process buyer_agent(1,www_serv_resp_1,www_serv_resp_2,
            www_buyer1_socket,www_buyer2_socket,buyer1_www_socket,
            buyer2_www_socket,www_shop_socket_1,www_shop_socket_2);
        buyer2: process buyer_agent(2,www_serv_resp_1,www_serv_resp_2,
            www_buyer1_socket,www_buyer2_socket,buyer1_www_socket,
            buyer2_www_socket,www_shop_socket_1,www_shop_socket_2);
        seller1 : process seller_agent (3,www_serv_resp_3,www_seller1_socket,
            www_shop_socket_3);
        www: process webServer(shop_www_socket_1,shop_www_socket_2,
            buyer1_www_socket,buyer2_www_socket,www_buyer1_socket,
            www_buyer2_socket,www_shop_socket_1,www_shop_socket_2,
            buyer1_op_req_1, buyer2_op_req_2, sh_user_session_1, sh_user_session_2,
            shop_www_socket_3,seller1_www_socket,www_seller1_socket,
            www_shop_socket_3,seller1_op_req_3);
        sh: process shop(www_shop_socket_1,www_shop_socket_2,www_serv_req_1,
            www_serv_req_2,www_shop_socket_3, www_serv_req_3,www_buyer1_socket,
            www_buyer2_socket,www_seller1_socket);
        it1 : process item(1,sh.item_id,www_serv_req_1,www_serv_req_2,www_serv_req_3
            );
        it2 : process item(2,sh.item_id,www_serv_req_1,www_serv_req_2,www_serv_req_3
            );


ASSIGN
```

```
        init (buyer1_www_socket) := 1;
        init (buyer2_www_socket) := 1;
        init (seller1_www_socket) := 1;
        init (shop_www_socket_1) := 0;
        init (shop_www_socket_2) := 0;
        init (shop_www_socket_3) := 0;
        init (www_shop_socket_1) := 0;
        init (www_shop_socket_2) := 0;
        init (www_shop_socket_3) := 0;
        init (www_buyer1_socket) := 0;
        init (www_buyer2_socket) := 0;
        init (www_seller1_socket) := 0;

DEFINE
        www_serv_resp_1:= www.serv_resp_1;
        www_serv_resp_2:= www.serv_resp_2;
        www_serv_resp_3:= www.serv_resp_3;
        www_serv_req_1:= www.serv_req_1;
        www_serv_req_2:= www.serv_req_2;
        www_serv_req_3:= www.serv_req_3;
        buyer1_op_req_1 := buyer1.op_req_1;
        buyer2_op_req_2 := buyer2.op_req_2;
        seller1_op_req_3  :=  seller1 .op_req_3;
        sh_user_session_1  := sh. user_session_1 ;
        sh_user_session_2  := sh. user_session_2 ;

-- Novas Especificacoes
SPEC AG !(sh.in_use = 1 & sh.in_use = 2)
SPEC EF (sh.in_use = 0)
SPEC EF (buyer1_op_req_1 = ho)
SPEC EF (buyer1_op_req_1 = sr)
SPEC EF (buyer1_op_req_1 = br)
SPEC EF (buyer1_op_req_1 = pa)
SPEC EF (buyer1_op_req_1 = di)
SPEC EF (buyer1_op_req_1 = ad)
SPEC EF (buyer1_op_req_1 = sl)
SPEC EF (!buyer1_www_socket= 0)
SPEC EF (!buyer1_www_socket= 1)
SPEC EF (buyer1_www_socket= 1 | buyer1_www_socket= 0)
SPEC EF !(www_shop_socket_1= 1)
SPEC EF(buyer1_www_socket = 1)
SPEC EF(buyer2_www_socket = 1)
SPEC EF(seller1_www_socket = 1)
SPEC EF(shop_www_socket_1 = 1)
SPEC EF(shop_www_socket_2 = 1)
SPEC EF(shop_www_socket_3 = 1)
SPEC EF(www_shop_socket_1 = 1)
SPEC EF(www_shop_socket_2 = 1)
SPEC EF(www_shop_socket_3 = 1)
SPEC EF(www_buyer1_socket = 1)
SPEC EF(www_buyer2_socket = 1)
SPEC EF(www_seller1_socket = 1)
SPEC EF(buyer1.op_resp_1 = sl)
SPEC EF(www_serv_resp_1 = sl_resp)
SPEC EF(www_serv_req_1 = sl_req)
SPEC EF(www_buyer1_socket=1 & www_serv_resp_1 = sl_resp)
SPEC AG(sh.user_session_1 = home) -> EF (sh.user_session_1 = home)
SPEC AG(sh.user_session_1 = home) -> EF (sh.user_session_1 = select)
SPEC AG(sh.user_session_1 = home) -> EF (sh.user_session_1 = search)
SPEC AG(sh.user_session_1 = home) -> EF (sh.user_session_1 = browse)

SPEC EF(buyer1.op_req_1 = ho)
SPEC EF(buyer1.op_req_1 = sl)
SPEC EF(buyer1.op_req_1 = sr)
SPEC EF(buyer1.op_req_1 = br)

SPEC EF (www.serv_req_1 = ho_req)
SPEC EF (www.serv_req_1 = sl_req)
SPEC EF (www.serv_req_1 = sr_req)
SPEC EF (www.serv_req_1 = br_req)

SPEC AG(sh.user_session_1 = home) -> EX (sh.user_session_1 = home)
SPEC AG(sh.user_session_1 = home) -> EX (sh.user_session_1 = select)
SPEC AG(sh.user_session_1 = home) -> EX (sh.user_session_1 = search)
SPEC AG(sh.user_session_1 = home) -> EX (sh.user_session_1 = browse)

SPEC EF (sh.user_session_1 = home & www_serv_req_1 = ho_req & www_shop_socket_1 =1)
SPEC EF (sh.user_session_1 = home & www_serv_req_1 = sl_req & www_shop_socket_1 =1)
SPEC EF (sh.user_session_1 = home & www_serv_req_1 = sr_req & www_shop_socket_1 =1)
SPEC EF (sh.user_session_1 = home & www_serv_req_1 = br_req & www_shop_socket_1 =1)

SPEC AG(sh.user_session_1 = home & www_serv_req_1 = ho_req & www_shop_socket_1 =1) -> EX (sh.user_session_1 = home)
SPEC AG(sh.user_session_1 = home & www_serv_req_1 = sl_req & www_shop_socket_1 =1) -> EX (sh.user_session_1 = select)
SPEC AG(sh.user_session_1 = home & www_serv_req_1 = sr_req & www_shop_socket_1 =1) -> EX (sh.user_session_1 = search)
SPEC AG(sh.user_session_1 = home & www_serv_req_1 = br_req & www_shop_socket_1 =1) -> EX (sh.user_session_1 = browse)

SPEC AG(sh.user_session_1 = search) -> EX (sh.user_session_1 = select)
SPEC AG(sh.user_session_1 = browse) -> EX (sh.user_session_1 = select)

SPEC EF (sh.user_session_1 = select)
SPEC EF !(shop_www_socket_1= 1)
SPEC EF !(www_buyer1_socket = 1)
SPEC AG((buyer1.op_resp_1 = ho & buyer1_www_socket=1) -> EF (buyer1_op_req_1 = ho))
SPEC AG((buyer1.op_resp_1 = ho & buyer1_www_socket=1) -> EF (buyer1_op_req_1 = sr))

SPEC EF (www_serv_req_1=ad_req)
SPEC EF (www_serv_req_2=ad_req)
SPEC EF (sh.in_use = 1)
SPEC EF (sh.in_use = 2)

-- Exemplos de CONSISTENCIA
SPEC AG !(sh.inventory > 2)
SPEC AG !(sh.inventory < 0)
SPEC AG (sh.inventory = 0) -> AX (sh.productIsAvailable = 0)
SPEC AG (sh.inventory > 0) -> AX (sh.productIsAvailable = 1)

--Exemplo de ATOMICIDADE
-- Existiam 2 itens, sendo (1 livre , 1 vendido)
-- Comprador pediu para comprar produto, entao se item tornar-se alocado,
    obrigatoriamente
-- a quantidade do produto deverah se tornar 0, ou entao nao foi atomico.
SPEC AG((it1.state=Available & it2.state=Reserved & buyer1.op_req_1=ad & !buyer2.
    op_req_2=ad) -> AX (it1.state=Reserved & sh.inventory = 0))

SPEC AG !( sh.user_session_1=select & EX (sh.user_session_1=pay))
SPEC AG !( sh.user_session_2=select & EX (sh.user_session_2=pay))

-- Modulo que representa o servidor de transacao do servidor de comercio eletronico

MODULE shop(www_shop_socket_1, www_shop_socket_2,www_serv_req_1,
    www_serv_req_2, www_shop_socket_3, www_serv_req_3, www_buyer1_socket,
    www_buyer2_socket,www_seller1_socket)

VAR
        user_session_1 : {home, select, search, browse, add, pay, error};
        user_session_2 : {home, select, search, browse, add, pay, error};
        shop_www_socket_1: boolean;
        shop_www_socket_2: boolean;
        shop_www_socket_3: boolean;
        in_use : 0..2;
        item_id  : 0..2;
        inventory : 0..2;
        bits_array : 0..3; -- cada item equivale a um bit
        productIsAvailable: boolean;

ASSIGN
        init ( user_session_1 ) := home;
        init ( user_session_2 ) := home;
        init (in_use) := 0;
        init ( bits_array ) := 3;
        init (item_id) := 0;
        init (inventory) := 2;
        init (productIsAvailable) := 1;

next(productIsAvailable) := case
        productIsAvailable = 0 & inventory > 0 : 1;
        productIsAvailable = 1 & inventory = 0 : 0;
        productIsAvailable = 0 & www_serv_req_1=ad_req : 0;
        productIsAvailable = 0 & www_serv_req_2=ad_req : 0;
        productIsAvailable = 0 & www_serv_req_1=di_req & inventory < 2 : 1;
        productIsAvailable = 0 & www_serv_req_2=di_req & inventory < 2 : 1;
        productIsAvailable = 0 & www_serv_req_1=pa_req : 0;
        productIsAvailable = 0 & www_serv_req_2=pa_req : 0;
        productIsAvailable = 1 & www_serv_req_1=br_req : 1 ;
        productIsAvailable = 1 & www_serv_req_2=br_req : 1 ;
        productIsAvailable = 1 & www_serv_req_1=ad_req & inventory = 1 & in_use
            = 1 : 0;
        productIsAvailable = 1 & www_serv_req_2=ad_req & inventory = 1 & in_use
            = 2 : 0;
        productIsAvailable = 1 & www_serv_req_1=di_req : 1;
        productIsAvailable = 1 & www_serv_req_2=di_req : 1;
        productIsAvailable = 1 & www_serv_req_1=pa_req : 1;
        productIsAvailable = 1 & www_serv_req_2=pa_req : 1;
        1: productIsAvailable;
esac;

next(inventory) := case
        www_serv_req_1 = ad_req & inventory > 1 & www_shop_socket_1=1: inventory
            - 1;
        www_serv_req_1 = ad_req & inventory = 1 & in_use=1 & www_shop_socket_1
            =1 : inventory - 1;
        www_serv_req_2 = ad_req & inventory > 1 & www_shop_socket_2=1: inventory
            - 1;
        www_serv_req_2 = ad_req & inventory = 1 & in_use=2 & www_shop_socket_2
            =1: inventory - 1;
        www_serv_req_1 = di_req & inventory < 2 & www_shop_socket_1=1: inventory
            + 1;
        www_serv_req_2 = di_req & inventory < 2 & www_shop_socket_2=1: inventory
            + 1;
        1: inventory;
```

```
esac;

next(bits_array) := case
        bits_array = 0 & www_serv_req_1 = di_req & www_shop_socket_1=1: bits_array
            + item_id;
        bits_array = 1 & www_serv_req_1 = di_req & item_id =2 & www_shop_socket_1
            =1: bits_array + item_id;
        bits_array = 2 & www_serv_req_1 = di_req & item_id =1 & www_shop_socket_1
            =1: bits_array + item_id;
        bits_array = 0 & www_serv_req_2 = di_req & www_shop_socket_2=1: bits_array
            + item_id;
        bits_array = 1 & www_serv_req_2 = di_req & item_id =2 & www_shop_socket_2
            =1: bits_array + item_id;
        bits_array = 2 & www_serv_req_2 = di_req & item_id =1 & www_shop_socket_2
            =1: bits_array + item_id;
        bits_array = 3 & www_serv_req_1 = ad_req & www_shop_socket_1=1: bits_array
            - item_id;
        bits_array = 2 & www_serv_req_1 = ad_req & item_id =2 & www_shop_socket_1
            =1: bits_array - item_id;
        bits_array = 1 & www_serv_req_1 = ad_req & item_id =1 & www_shop_socket_1
            =1: bits_array - item_id;
        bits_array = 3 & www_serv_req_2 = ad_req & www_shop_socket_2=1: bits_array
            - item_id;
        bits_array = 2 & www_serv_req_2 = ad_req & item_id =2 & www_shop_socket_2
            =1: bits_array - item_id;
        bits_array = 1 & www_serv_req_2 = ad_req & item_id =1 & www_shop_socket_2
            =1: bits_array - item_id;
        1: bits_array;
esac;

next(item_id) := case
        bits_array = 0 : item_id = 0; -- ãno áser feita çãao sobre itens
        bits_array = 1 : item_id = 1; -- çãao áser realizada sobre item 1
        bits_array = 2 : item_id = 2; -- çãao áser realizada sobre item 2
        bits_array = 3 : item_id = {1,2}; -- çãao áser realizada sobre item 1 ou 2
        1: item_id;
esac;

```

```
next(in_use) := case
          in_use = 0 & www_serv_req_1=ad_req & www_serv_req_2=ad & inventory
               <= 2: {1,2};
          in_use = 0 & www_serv_req_1=ad_req & inventory <= 2: 1;
          in_use = 0 & www_serv_req_2=ad_req & inventory <= 2: 2;
          in_use = 1 & www_serv_req_1=di_req : 0;
          in_use = 1 & www_serv_req_1=pa_req : 0;
          in_use = 2 & www_serv_req_2=pa_req : 0;
          in_use = 2 & www_serv_req_2=di_req : 0;
          1: in_use;
esac;

next(user_session_1) := case
          user_session_1 = home & www_serv_req_1=ho_req & www_shop_socket_1=1 :
              home;
          user_session_1 = home & www_serv_req_1=sl_req & www_shop_socket_1=1 :
              select;
          user_session_1 = home & www_serv_req_1=sr_req & www_shop_socket_1=1 :
              search;
          user_session_1 = home & www_serv_req_1=br_req & www_shop_socket_1=1 :
              browse;
          user_session_1 = select & www_serv_req_1=ho_req & www_shop_socket_1=1 :
              home;
          user_session_1 = select & www_serv_req_1=sr_req & www_shop_socket_1=1 :
              search;
          user_session_1 = select & www_serv_req_1=ad_req & www_shop_socket_1=1 &
              next(productIsAvailable)=1 & in_use=0: add;
          user_session_1 = select & www_serv_req_1=ad_req & www_shop_socket_1=1 &
              next(productIsAvailable)=1 & in_use=1: add;
          user_session_1 = select & www_serv_req_1=ad_req & www_shop_socket_1=1 &
              next(productIsAvailable)=0 : error;
          user_session_1 = search & www_serv_req_1=sr_req & www_shop_socket_1=1 :
              search;
          user_session_1 = search & www_serv_req_1=ho_req & www_shop_socket_1=1 :
              home;
          user_session_1 = search & www_serv_req_1=sl_req & www_shop_socket_1=1 :
              select;
          user_session_1 = search & www_serv_req_1=br_req & www_shop_socket_1=1 :
            browse;
        user_session_1 = browse & www_serv_req_1=br_req & www_shop_socket_1=1 :
            browse;
        user_session_1 = browse & www_serv_req_1=ho_req & www_shop_socket_1=1 :
            home;
        user_session_1 = browse & www_serv_req_1=sr_req & www_shop_socket_1=1 :
            search;
        user_session_1 = browse & www_serv_req_1=sl_req & www_shop_socket_1=1 :
            select;
        user_session_1 = add & www_serv_req_1=pa_req & www_shop_socket_1=1 : pay;
        user_session_1 = add & www_serv_req_1=di_req & www_shop_socket_1=1 :
            home;
        user_session_1 = add & www_serv_req_1=sr_req & www_shop_socket_1=1 :
            search;
        user_session_1 = add & www_serv_req_1=br_req & www_shop_socket_1=1 :
            browse;
        user_session_1 = pay & www_serv_req_1=ho_req & www_shop_socket_1=1 :
            home;
        user_session_1 = error: home;
        1: user_session_1 ;
esac;

next(shop_www_socket_1) := case
        shop_www_socket_1 = 0 & www_shop_socket_1 = 1: 1;
        shop_www_socket_1 = 1 & www_buyer1_socket = 1: 0;
        1: shop_www_socket_1;
esac;

next(user_session_2) := case
        user_session_2 = home & www_serv_req_2=ho_req & www_shop_socket_2=1 :
            home;
        user_session_2 = home & www_serv_req_2=sl_req & www_shop_socket_2=1 :
            select;
        user_session_2 = home & www_serv_req_2=sr_req & www_shop_socket_2=1 :
            search;
        user_session_2 = home & www_serv_req_2=br_req & www_shop_socket_2=1 :
            browse;
        user_session_2 = select & www_serv_req_2=ho_req & www_shop_socket_2=1 :
            home;
        user_session_2 = select & www_serv_req_2=sr_req & www_shop_socket_2=1 :
            search;
        user_session_2 = select & www_serv_req_2=ad_req & www_shop_socket_2=1 &
            next(productIsAvailable)=1 & in_use=0: add;
        user_session_2 = select & www_serv_req_2=ad_req & www_shop_socket_2=1 &
            next(productIsAvailable)=1 & in_use=2: add;
        user_session_2 = select & www_serv_req_2=ad_req & www_shop_socket_2=1 &
            next(productIsAvailable)=0 : error;
        user_session_2 = search & www_serv_req_2=sr_req & www_shop_socket_2=1 :
            search;
        user_session_2 = search & www_serv_req_2=ho_req & www_shop_socket_2=1 :
            home;
        user_session_2 = search & www_serv_req_2=sl_req & www_shop_socket_2=1 :
            select;
        user_session_2 = search & www_serv_req_2=br_req & www_shop_socket_2=1 :
            browse;
        user_session_2 = browse & www_serv_req_2=br_req & www_shop_socket_2=1 :
            browse;
        user_session_2 = browse & www_serv_req_2=ho_req & www_shop_socket_2=1 :
            home;
        user_session_2 = browse & www_serv_req_2=sr_req & www_shop_socket_2=1 :
            search;
        user_session_2 = browse & www_serv_req_2=sl_req & www_shop_socket_2=1 :
            select;
        user_session_2 = add & www_serv_req_2=pa_req & www_shop_socket_2=1 : pay;
        user_session_2 = add & www_serv_req_2=di_req & www_shop_socket_2=1 :
            home;
        user_session_2 = add & www_serv_req_2=sr_req & www_shop_socket_2=1 :
            search;
        user_session_2 = add & www_serv_req_2=br_req & www_shop_socket_2=1 :
            browse;
        user_session_2 = pay & www_serv_req_2=ho_req & www_shop_socket_2=1 :
            home;
        user_session_2 = error: home;
        1: user_session_2 ;
    esac;

```

```
next(shop_www_socket_2) := case
          shop_www_socket_2 = 0 & www_shop_socket_2 = 1: 1;
          shop_www_socket_2 = 1 & www_buyer2_socket = 1: 0;
          1: shop_www_socket_2;
esac;

next(shop_www_socket_3) := case
          shop_www_socket_3 = 0 & www_shop_socket_3 = 1: 1;
          shop_www_socket_3 = 1 & www_seller1_socket = 1: 0;
          1: shop_www_socket_3;
esac;

FAIRNESS running


-- Modulo que representa o item, objeto alvo das operacoes do agente comprador

MODULE item(id,iditem,www_serv_req_1, www_serv_req_2, www_serv_req_3)

VAR
          state : {Reserved, Sold, Available};

ASSIGN
          init(state) := Available; -- coloque isso como padrao

next(state) := case
          state = Available & www_serv_req_3=ds_req : Available;
          state = Available & www_serv_req_3=ch_req : Available;
          state = Reserved & www_serv_req_3=ds_req : Reserved;
          state = Reserved & www_serv_req_3=ch_req : Reserved;
          state = Sold & www_serv_req_3=ds_req : Available;
          state = Sold & www_serv_req_3=ch_req : Sold;
          state = Available & www_serv_req_1=br_req : Available;
          state = Available & www_serv_req_1=ad_req & id = iditem: Reserved;
          state = Available & www_serv_req_1=di_req & id = iditem: Available;
          state = Available & www_serv_req_1=pa_req : Available;
          state = Reserved & www_serv_req_1=br_req : Reserved ;
          state = Reserved & www_serv_req_1=ad_req & id = iditem: Reserved;
```

```
                state = Reserved & www_serv_req_1=di_req & id = iditem: Available;
                state = Reserved & www_serv_req_1=pa_req & id = iditem: Sold;
                state = Sold & www_serv_req_1=br_req : Sold;
                state = Sold & www_serv_req_1=ad_req : Sold;
                state = Sold & www_serv_req_1=di_req : Sold;
                state = Sold & www_serv_req_1=pa_req : Sold;
                state = Available & www_serv_req_2=br_req : Available;
                state = Available & www_serv_req_2=ad_req & id = iditem: Reserved;
                state = Available & www_serv_req_2=di_req & id = iditem: Available;
                state = Available & www_serv_req_2=pa_req : Available;
                state = Reserved & www_serv_req_2=br_req : Reserved ;
                state = Reserved & www_serv_req_2=ad_req & id = iditem: Reserved;
                state = Reserved & www_serv_req_2=di_req & id = iditem: Available;
                state = Reserved & www_serv_req_2=pa_req & id = iditem: Sold;
                state = Sold & www_serv_req_2=br_req : Sold;
                state = Sold & www_serv_req_2=ad_req : Sold;
                state = Sold & www_serv_req_2=di_req : Sold;
                state = Sold & www_serv_req_2=pa_req : Sold;
                1: state;
    esac;

    FAIRNESS running


    -- Modulo que representa o servidor WWW do servidor de comercio eletronico

    MODULE webServer(shop_www_socket_1,shop_www_socket_2,buyer1_www_socket,
        buyer2_www_socket,www_buyer1_socket,www_buyer2_socket,www_shop_socket_1,
        www_shop_socket_2,buyer1_op_req_1, buyer2_op_req_2, sh_user_session_1,
        sh_user_session_2, shop_www_socket_3, seller1_www_socket, www_seller1_socket,
        www_shop_socket_3, seller1_op_req_3)

    VAR
            serv_req_1 : {ho_req, sl_req , sr_req , br_req , ad_req, di_req , pa_req};
            serv_resp_1 : {ho_resp, sl_resp , sr_resp , br_resp , ad_resp, di_resp , pa_resp ,
                er_resp};
            serv_req_2 : {ho_req, sl_req , sr_req , br_req , ad_req, di_req , pa_req};
            serv_resp_2 : {ho_resp, sl_resp , sr_resp , br_resp , ad_resp, di_resp , pa_resp ,
                er_resp};
            serv_req_3 : {ds_req, ch_req};
            serv_resp_3 : {ds_resp, ch_resp};

ASSIGN
            init (serv_req_1) := ho_req;
            init (serv_req_2) := ho_req;
            init (serv_req_3) := ch_req;
            init (serv_resp_1) := ho_resp;
            init (serv_resp_2) := ho_resp;
            init (serv_resp_3) := ch_resp;

next(serv_req_1) := case
            buyer1_op_req_1 = ho & buyer1_www_socket=1 : ho_req;
            buyer1_op_req_1 = sl & buyer1_www_socket=1 : sl_req;
            buyer1_op_req_1 = sr & buyer1_www_socket=1 : sr_req;
            buyer1_op_req_1 = br & buyer1_www_socket=1 : br_req;
            buyer1_op_req_1 = ad & buyer1_www_socket=1 : ad_req;
            buyer1_op_req_1 = di & buyer1_www_socket=1 : di_req;
            buyer1_op_req_1 = pa & buyer1_www_socket=1 : pa_req;
            1: serv_req_1;
esac;

next(www_shop_socket_1) := case
            www_shop_socket_1 = 0 & buyer1_www_socket = 1: 1;
            www_shop_socket_1 = 1 & shop_www_socket_1 = 1 : 0;
            1: www_shop_socket_1;
esac;

next(serv_req_2) := case
            buyer2_op_req_2 = ho & buyer2_www_socket=1 : ho_req;
            buyer2_op_req_2 = sl & buyer2_www_socket=1 : sl_req;
            buyer2_op_req_2 = sr & buyer2_www_socket=1 : sr_req;
            buyer2_op_req_2 = br & buyer2_www_socket=1 : br_req;
            buyer2_op_req_2 = ad & buyer2_www_socket=1 : ad_req;
            buyer2_op_req_2 = di & buyer2_www_socket=1 : di_req;
            buyer2_op_req_2 = pa & buyer2_www_socket=1 : pa_req;
            1: serv_req_2;
```

```
esac;

next(www_shop_socket_2) := case
        www_shop_socket_2 = 0 & buyer2_www_socket = 1: 1;
        www_shop_socket_2 = 1 & shop_www_socket_2 = 1 : 0;
        1: www_shop_socket_2;
esac;


next(serv_resp_1) := case
        next(sh_user_session_1) = home & shop_www_socket_1=1 : ho_resp;
        next(sh_user_session_1) = select & shop_www_socket_1=1 : sl_resp;
        next(sh_user_session_1) = search & shop_www_socket_1=1 : sr_resp;
        next(sh_user_session_1) = browse & shop_www_socket_1=1 : br_resp;
        next(sh_user_session_1) = home & shop_www_socket_1=1 : ho_resp;
        next(sh_user_session_1) = search & shop_www_socket_1=1 : sr_resp;
        next(sh_user_session_1) = add & shop_www_socket_1=1 : ad_resp;
        next(sh_user_session_1) = add & shop_www_socket_1=1 : ad_resp;
        next(sh_user_session_1) = error & shop_www_socket_1=1 : er_resp;
        next(sh_user_session_1) = search & shop_www_socket_1=1 : sr_resp;
        next(sh_user_session_1) = home & shop_www_socket_1=1 : ho_resp;
        next(sh_user_session_1) = select & shop_www_socket_1=1 : sl_resp;
        next(sh_user_session_1) = browse & shop_www_socket_1=1 : br_resp;
        next(sh_user_session_1) = browse & shop_www_socket_1=1 : br_resp;
        next(sh_user_session_1) = home & shop_www_socket_1=1 : ho_resp;
        next(sh_user_session_1) = search & shop_www_socket_1=1 : sr_resp;
        next(sh_user_session_1) = select & shop_www_socket_1=1 : sl_resp;
        next(sh_user_session_1) = pay & shop_www_socket_1=1 : pa_resp;
        next(sh_user_session_1) = home & shop_www_socket_1=1 : ho_resp;
        next(sh_user_session_1) = search & shop_www_socket_1=1 : sr_resp;
        next(sh_user_session_1) = browse & shop_www_socket_1=1 : br_resp;
        next(sh_user_session_1) = home & shop_www_socket_1=1 : ho_resp;
        next(sh_user_session_1) = home & shop_www_socket_1=1 : ho_resp;
        1: serv_resp_1;
esac;

--next(serv_resp_1) := case
--      sh_user_session_1 = home & next(sh_user_session_1) = home &
```

```
        shop_www_socket_1=1 : ho_resp;
444 --   sh_user_session_1 = home & next(sh_user_session_1) = select &
        shop_www_socket_1=1 : sl_resp;
445 --   sh_user_session_1 = home & next(sh_user_session_1) = search &
        shop_www_socket_1=1 : sr_resp;
446 --   sh_user_session_1 = home & next(sh_user_session_1) = browse &
        shop_www_socket_1=1 : br_resp;
447 --   sh_user_session_1 = select & next(sh_user_session_1) = home &
        shop_www_socket_1=1 : ho_resp;
448 --   sh_user_session_1 = select & next(sh_user_session_1) = search &
        shop_www_socket_1=1 : sr_resp;
449 --   sh_user_session_1 = select & next(sh_user_session_1) = add &
        shop_www_socket_1=1 : ad_resp;
450 --   sh_user_session_1 = select & next(sh_user_session_1) = add &
        shop_www_socket_1=1 : ad_resp;
451 --   sh_user_session_1 = select & next(sh_user_session_1) = error &
        shop_www_socket_1=1 : er_resp;
452 --   sh_user_session_1 = search & next(sh_user_session_1) = search &
        shop_www_socket_1=1 : sr_resp;
453 --   sh_user_session_1 = search & next(sh_user_session_1) = home &
        shop_www_socket_1=1 : ho_resp;
454 --   sh_user_session_1 = search & next(sh_user_session_1) = select &
        shop_www_socket_1=1 : sl_resp;
455 --   sh_user_session_1 = search & next(sh_user_session_1) = browse &
        shop_www_socket_1=1 : br_resp;
456 --   sh_user_session_1 = browse & next(sh_user_session_1) = browse &
        shop_www_socket_1=1 : br_resp;
457 --   sh_user_session_1 = browse & next(sh_user_session_1) = home &
        shop_www_socket_1=1 : ho_resp;
458 --   sh_user_session_1 = browse & next(sh_user_session_1) = search &
        shop_www_socket_1=1 : sr_resp;
459 --   sh_user_session_1 = browse & next(sh_user_session_1) = select &
        shop_www_socket_1=1 : sl_resp;
460 --   sh_user_session_1 = add & next(sh_user_session_1) = pay & shop_www_socket_1
        =1 : pa_resp;
461 --   sh_user_session_1 = add & next(sh_user_session_1) = home & shop_www_socket_1
        =1 : ho_resp;
462 --   sh_user_session_1 = add & next(sh_user_session_1) = search &
```

```
        shop_www_socket_1=1 : sr_resp;
--    sh_user_session_1  = add & next(sh_user_session_1) = browse &
        shop_www_socket_1=1 : br_resp;
--    sh_user_session_1  = pay & next(sh_user_session_1) = home & shop_www_socket_1
        =1 : ho_resp;
--    sh_user_session_1  = error & next(sh_user_session_1) = home &
        shop_www_socket_1=1 : ho_resp;
--    1: serv_resp_1;
--esac;


next(www_buyer1_socket) := case
        www_buyer1_socket = 0 & shop_www_socket_1 = 1: 1;
        www_buyer1_socket = 1 & buyer1_www_socket = 1 : 0;
        1: www_buyer1_socket;
esac;

next(serv_resp_2) := case
        next(sh_user_session_2) = home & shop_www_socket_2=1 : ho_resp;
        next(sh_user_session_2) = select & shop_www_socket_2=1 : sl_resp;
        next(sh_user_session_2) = search & shop_www_socket_2=1 : sr_resp;
        next(sh_user_session_2) = browse & shop_www_socket_2=1 : br_resp;
        next(sh_user_session_2) = home & shop_www_socket_2=1 : ho_resp;
        next(sh_user_session_2) = search & shop_www_socket_2=1 : sr_resp;
        next(sh_user_session_2) = add & shop_www_socket_2=1 : ad_resp;
        next(sh_user_session_2) = add & shop_www_socket_2=1 : ad_resp;
        next(sh_user_session_2) = error & shop_www_socket_2=1 : er_resp;
        next(sh_user_session_2) = search & shop_www_socket_2=1 : sr_resp;
        next(sh_user_session_2) = home & shop_www_socket_2=1 : ho_resp;
        next(sh_user_session_2) = select & shop_www_socket_2=1 : sl_resp;
        next(sh_user_session_2) = browse & shop_www_socket_2=1 : br_resp;
        next(sh_user_session_2) = browse & shop_www_socket_2=1 : br_resp;
        next(sh_user_session_2) = home & shop_www_socket_2=1 : ho_resp;
        next(sh_user_session_2) = search & shop_www_socket_2=1 : sr_resp;
        next(sh_user_session_2) = select & shop_www_socket_2=1 : sl_resp;
        next(sh_user_session_2) = pay & shop_www_socket_2=1 : pa_resp;
        next(sh_user_session_2) = home & shop_www_socket_2=1 : ho_resp;
        next(sh_user_session_2) = search & shop_www_socket_2=1 : sr_resp;
        next( sh_user_session_2 ) = browse & shop_www_socket_2=1 : br_resp;
        next( sh_user_session_2 ) = home & shop_www_socket_2=1 : ho_resp;
        next( sh_user_session_2 ) = home & shop_www_socket_2=1 : ho_resp;
        1: serv_resp_2;
esac;

--next(serv_resp_2) := case
--    sh_user_session_2 = home & next(sh_user_session_2) = home &
    shop_www_socket_2=1 : ho_resp;
--    sh_user_session_2 = home & next(sh_user_session_2) = select &
    shop_www_socket_2=1 : sl_resp;
--    sh_user_session_2 = home & next(sh_user_session_2) = search &
    shop_www_socket_2=1 : sr_resp;
--    sh_user_session_2 = home & next(sh_user_session_2) = browse &
    shop_www_socket_2=1 : br_resp;
--    sh_user_session_2 = select & next(sh_user_session_2 ) = home &
    shop_www_socket_2=1 : ho_resp;
--    sh_user_session_2 = select & next(sh_user_session_2 ) = search &
    shop_www_socket_2=1 : sr_resp;
--    sh_user_session_2 = select & next(sh_user_session_2 ) = add &
    shop_www_socket_2=1 : ad_resp;
--    sh_user_session_2 = select & next(sh_user_session_2 ) = add &
    shop_www_socket_2=1 : ad_resp;
--    sh_user_session_2 = select & next(sh_user_session_2 ) = error &
    shop_www_socket_2=1 : er_resp;
--    sh_user_session_2 = search & next(sh_user_session_2) = search &
    shop_www_socket_2=1 : sr_resp;
--    sh_user_session_2 = search & next(sh_user_session_2) = home &
    shop_www_socket_2=1 : ho_resp;
--    sh_user_session_2 = search & next(sh_user_session_2) = select &
    shop_www_socket_2=1 : sl_resp;
--    sh_user_session_2 = search & next(sh_user_session_2) = browse &
    shop_www_socket_2=1 : br_resp;
--    sh_user_session_2 = browse & next(sh_user_session_2) = browse &
    shop_www_socket_2=1 : br_resp;
--    sh_user_session_2 = browse & next(sh_user_session_2) = home &
    shop_www_socket_2=1 : ho_resp;
--    sh_user_session_2 = browse & next(sh_user_session_2) = search &
```

```
        shop_www_socket_2=1 : sr_resp;
--   sh_user_session_2  = browse & next(sh_user_session_2) = select &
        shop_www_socket_2=1 : sl_resp;
--   sh_user_session_2  = add & next(sh_user_session_2) = pay & shop_www_socket_2
        =1 : pa_resp;
--   sh_user_session_2  = add & next(sh_user_session_2) = home & shop_www_socket_2
        =1 : ho_resp;
--   sh_user_session_2  = add & next(sh_user_session_2) = search &
        shop_www_socket_2=1 : sr_resp;
--   sh_user_session_2  = add & next(sh_user_session_2) = browse &
        shop_www_socket_2=1 : br_resp;
--   sh_user_session_2  = pay & next(sh_user_session_2) = home & shop_www_socket_2
        =1 : ho_resp;
--   sh_user_session_2  = error & next(sh_user_session_2) = home &
        shop_www_socket_2=1 : ho_resp;
--   1: serv_resp_2;
--esac;

next(www_buyer2_socket) := case
        www_buyer2_socket = 0 & shop_www_socket_2 = 1: 1;
        www_buyer2_socket = 1 & buyer2_www_socket = 1 : 0;
        1: www_buyer2_socket;
esac;

next(serv_req_3) := case
        seller1_op_req_3  = ds & seller1_www_socket=1 : {ds_req};
        seller1_op_req_3  = ch & seller1_www_socket=1 : {ch_req};
        1: serv_req_3;
esac;

next(www_shop_socket_3) := case
        www_shop_socket_3 = 0 & seller1_www_socket = 1: 1;
        www_shop_socket_3 = 1 & shop_www_socket_3 = 1 : 0;
        1: www_shop_socket_3;
esac;

-- nao ha necessidade de controle de sessao do vendedor

```

```
next(www_seller1_socket) := case
          www_seller1_socket = 0 & shop_www_socket_3 = 1: 1;
          www_seller1_socket = 1 & seller1_www_socket = 1 : 0;
          1: www_seller1_socket;
esac;


FAIRNESS running

-- Modulo que representa o usuario, o agente comprador

MODULE buyer_agent(id,www_serv_resp_1,www_serv_resp_2,www_buyer1_socket,
    www_buyer2_socket,buyer1_www_socket,buyer2_www_socket,www_shop_socket_1,
    www_shop_socket_2)

VAR
          op_req_1: {ho, sl, sr, br, ad, di, pa};
          op_req_2: {ho, sl, sr, br, ad, di, pa};
          op_resp_1: {ho, sl, sr, br, ad, di, pa, er};
          op_resp_2: {ho, sl, sr, br, ad, di, pa, er};

ASSIGN
          init (op_req_1) := ho;
          init (op_req_2) := ho;
          init (op_resp_1) := ho;
          init (op_resp_2) := ho;

next(op_req_1) := case
          op_resp_1 = ho & id=1 & buyer1_www_socket=1: {ho,sl,sr,br};
          op_resp_1 = sl & id=1 & buyer1_www_socket=1: {ho,ad,sr};
          op_resp_1 = sr & id=1 & buyer1_www_socket=1: {sl,ho,br,sr};
          op_resp_1 = br & id=1 & buyer1_www_socket=1: {ho,br,sr,sl};
          op_resp_1 = ad & id=1 & buyer1_www_socket=1: {sr,pa,br,di};
          op_resp_1 = pa & id=1 & buyer1_www_socket=1: {ho};
          op_resp_1 = er & id=1 & buyer1_www_socket=1: {ho};
          1: op_req_1;
esac;

```

```
next(buyer1_www_socket) := case
          op_resp_1 = ho & id=1 & buyer1_www_socket=1 & www_shop_socket_1 =1 :
              buyer1_www_socket=0;
          op_resp_1 = sl & id=1 & buyer1_www_socket=1 & www_shop_socket_1 =1 :
              buyer1_www_socket=0;
          op_resp_1 = sr & id=1 & buyer1_www_socket=1 & www_shop_socket_1 =1 :
              buyer1_www_socket=0;
          op_resp_1 = br & id=1 & buyer1_www_socket=1 & www_shop_socket_1 =1 :
              buyer1_www_socket=0;
          op_resp_1 = ad & id=1 & buyer1_www_socket=1 & www_shop_socket_1 =1 :
              buyer1_www_socket=0;
          op_resp_1 = pa & id=1 & buyer1_www_socket=1 & www_shop_socket_1 =1 :
              buyer1_www_socket=0;
          op_resp_1 = er & id=1 & buyer1_www_socket=1 & www_shop_socket_1 =1 :
              buyer1_www_socket=0;
          www_buyer1_socket = 1 : buyer1_www_socket = 1;
          1: buyer1_www_socket;
esac;


next(op_req_2) := case
          op_resp_2 = ho & id=2 & buyer2_www_socket=1: {ho,sl,sr,br};
          op_resp_2 = sl & id=2 & buyer2_www_socket=1: {ho,ad,sr};
          op_resp_2 = sr & id=2 & buyer2_www_socket=1: {sl,ho,br,sr};
          op_resp_2 = br & id=2 & buyer2_www_socket=1: {ho,br,sr,sl};
          op_resp_2 = ad & id=2 & buyer2_www_socket=1: {sr,pa,br,di};
          op_resp_2 = pa & id=2 & buyer2_www_socket=1: {ho};
          op_resp_2 = er & id=2 & buyer2_www_socket=1: {ho};
          1: op_req_2;
esac;

next(buyer2_www_socket) := case
          op_resp_1 = ho & id=2 & buyer2_www_socket=1 & www_shop_socket_2 =1 :
              buyer2_www_socket=0;
          op_resp_1 = sl & id=2 & buyer2_www_socket=1 & www_shop_socket_2 =1 :
              buyer2_www_socket=0;
          op_resp_1 = sr & id=2 & buyer2_www_socket=1 & www_shop_socket_2 =1 :
              buyer2_www_socket=0;
```

```
            op_resp_1 = br & id=2 & buyer2_www_socket=1 & www_shop_socket_2 =1 : buyer2_www_socket=0;
            op_resp_1 = ad & id=2 & buyer2_www_socket=1 & www_shop_socket_2 =1 : buyer2_www_socket=0;
            op_resp_1 = pa & id=2 & buyer2_www_socket=1 & www_shop_socket_2 =1 : buyer2_www_socket=0;
            op_resp_1 = er & id=2 & buyer2_www_socket=1 & www_shop_socket_2 =1 : buyer2_www_socket=0;
            www_buyer2_socket = 1 : buyer2_www_socket = 1;
            1: buyer2_www_socket;
esac;


next(op_resp_1) := case
            www_buyer1_socket=1 & www_serv_resp_1 = ho_resp : ho;
            www_buyer1_socket=1 & www_serv_resp_1 = sl_resp : sl;
            www_buyer1_socket=1 & www_serv_resp_1 = sr_resp : sr;
            www_buyer1_socket=1 & www_serv_resp_1 = br_resp : br;
            www_buyer1_socket=1 & www_serv_resp_1 = ad_resp : ad;
            www_buyer1_socket=1 & www_serv_resp_1 = pa_resp : pa;
            www_buyer1_socket=1 & www_serv_resp_1 = er_resp : er;
            1: op_resp_1;
esac;

next(op_resp_2) := case
            www_buyer2_socket=1 & www_serv_resp_2 = ho_resp : ho;
            www_buyer2_socket=1 & www_serv_resp_2 = sl_resp : sl;
            www_buyer2_socket=1 & www_serv_resp_2 = sr_resp : sr;
            www_buyer2_socket=1 & www_serv_resp_2 = br_resp : br;
            www_buyer2_socket=1 & www_serv_resp_2 = ad_resp : ad;
            www_buyer2_socket=1 & www_serv_resp_2 = pa_resp : pa;
            www_buyer2_socket=1 & www_serv_resp_2 = er_resp : er;
            1: op_resp_2;
esac;

FAIRNESS running

-- Modulo que representa o usuario, o agente vendedor
```

```

MODULE seller_agent(id,www_serv_resp_3,www_seller1_socket,www_shop_socket_3)

VAR
          op_req_3: {ds, ch, no}; -- ds = disponibilizar; ch = alterar; no = none
          op_resp_3: {ho,er};
          seller1_www_socket: boolean;

ASSIGN
          init(op_req_3) := no;
          init(op_resp_3) := ho;

next(op_req_3) := case
          op_resp_3 = ho & id=3 & seller1_www_socket=1: {ds,ch,no};
          op_resp_3 = er & id=3 & seller1_www_socket=1: {ds,ch,no};
          1: op_req_3;
esac;

next(seller1_www_socket) := case
          op_resp_3 = ho & id=3 & seller1_www_socket=1 & www_shop_socket_3 =1 :
              seller1_www_socket=0;
          op_resp_3 = er & id=3 & seller1_www_socket=3 & www_shop_socket_3 =1 :
              seller1_www_socket=0;
          www_seller1_socket = 1 : seller1_www_socket = 1;
          1: seller1_www_socket;
esac;

next(op_resp_3) := case
          www_seller1_socket=1 & www_serv_resp_3 = ho_resp : ho;
          www_seller1_socket=1 & www_serv_resp_3 = er_resp : er;
          1: op_resp_3;
esac;

FAIRNESS running
```

Glossary

These explanations are provided to help the nonspecialist. They are intended to reflect the technical uses of the terms considered, but do not attempt to incorporate subtleties that concern the specialist.

Abstraction: the process of simplifying certain details of a system description or model so that the main issues are exposed. Abstraction is the key to gaining intellectual mastery of any complex system, and a prerequisite to effective use of formal methods. It requires great skill and experience to use abstraction to best effect.

In formal methods, abstraction is part of the process of developing a mathematical model that is a simplification or approximation of reality but that retains the properties of interest. In physics, for example, it is customary to model a moving object as a point mass, and to ignore its shape. Similarly in the case of a flight-control system, one can analyze properties of, say, the clock synchronization algorithm or the redundancy management mechanisms by abstracting these away from the larger and more complex system in which they are embedded.

Correctness: the property that a system does what it is expected and required to do. Formal methods cannot establish correctness in this most general sense because they deal with formal models of the system that may be inaccurate or incomplete, and with formal statements of requirements that may not capture all expectations. The difference between the real and modeled worlds is a potential source of error that attends all uses of mathematical modeling in engineering (e.g., in numerical aerodynamics or stress calculations) and that must be controlled by validating the models concerned. The difference between expectations and documented requirements is another problem that attends all engineering activities. Formal methods provide ways to make the specifications of assumptions and requirements precise; formal validation (q.v.) can then be used to ensure that the specifications are adequately complete and correct.

Correctness does not ensure safety or other critical properties, since the system requirements and expectations may not address these issues (correctly or at completely). System requirements usually describe functional properties (i.e., what the system is to do); it is necessary to establish nonfunctional properties such as safety and security (which often describe what the system is not to do) by separate scrutiny (based, e.g., on hazard analysis, or threat analysis). Formal methods can be used in these processes.

Design Faults: mistakes in the design of the system, or in the understanding of its requirements and assumptions, that cause it to do the wrong thing or to fail in certain circumstances. Also called generic faults. Modular redundancy provides no protection against these faults.

Formal logic: symbolic notation equipped with rules for constructing formal proofs. Formal logic consists of a language for writing statements and syntactic rules of inference for constructing proofs using these statements. Formal logic supports a form of reasoning that does not rely on the subjective interpretation of the symbols used.

There are many formal logics; they differ in what concepts they can express, and in how difficult it is to discover or check proofs. Propositional logic, first order logic, higher-order logic, the simple theory of types, and temporal logic are all examples of formal logics that find application in formal methods. These logics are generally augmented with certain theories defined within them that provide definitions or axiomatizations for useful mathematical concepts, such as sets, numbers, state machines, etc.

Formal Proof and formal deduction: Formal deduction is the process of deriving a sentence expressed in a formal logic from others through application of one or more rules of inference.

A formal proof is a demonstration that a given sentence (the theorem) follows by formal deduction from given (i.e., assumed) sentences called premises.

Formal methods: methods that use ideas and techniques from mathematical or formal logic (q.v.) to specify and reason about computational systems (both hardware and software).

Formal specification: a description of some computational system expressed in a notation based on formal logic. Generally, the specification states certain assumptions about the context in which the system is to operate (e.g., laws of physics, properties of subsystems and of systems with which the given system is to interact), and certain properties required of the system. A requirements specification need specify no more than this; a design specification will specify some elements of how the desired properties are to be achieved–e.g., algorithms and decomposition into subsystems.

Formal validation: a process for gaining confidence that top-level formal specifications of requirements and assumptions are correct. Formal verification (q.v.) cannot be applied at these levels because there are no higher-level requirements or more basic assumptions against which to verify them: processes of review and examination must be used instead. Formal validation consists of challenging the formal specifications by proposing and attempting to prove theorems that ought to follow from them (i.e., "if I've got this right, then this ought to follow.")

Formal verification: the process of showing, by means of formal deduction, that a formal design specification satisfies its formal requirements specification. The formal description of a design and its assumptions supply the premises, and the requirements supply the theorem to be proved. In hierarchical developments, assumptions and designs at one level become requirements at another, so the formal verification process can be repeated through many levels of design and abstraction. At the topmost level, validation (q.v.) must be employed.

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. The technique has been applied to several complex industrial systems such as the Futurebus+ [22] and the PCI local bus protocols [14].

Theorem proving and proof checking: Given a putative theorem and its premises, a theorem prover attempts to discover a proof that the theorem follows from the premises; on the other hand, a proof checker simply checks that a given proof is valid according to the rules of deduction for the logic concerned. Both these processes can be automated. A theorem prover is a computer program that uses search, heuristics, and user-supplied hints to guide its attempt to discover a proof. A proof checker is a computer program that is used interactively: a human user proposes proof steps and the proof checker checks they are valid and carries them out. The most effective automated assistance for formal methods is generally obtained by a hybrid combination of these approaches: the user proposes fairly big steps and the proof checker uses theorem proving techniques to fill in the gaps and take care of the details. Examples of theorem provers include Otter, Nqthm, PTTP, RRL, and TPS. Examples of proof checkers include Automath, Coq, HOL, Isabelle, and Nuprl. Hybrids include Eves, IMPS, PC-Nqthm, and PVS. Other forms of automated analysis that can be applied to formal specifications include model checking, language inclusion, and state exploration; examples of systems that perform these analysis are SMV [67], NuSMV [17, 18], COSPAN [81], Verus [90], and Murϕ [27].