Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 109 (2004) 125–136
www.elsevier.com/locate/entcs

Verifying Object-Based Graph Grammars[1]

Osmar Marchi dos Santos[2], Fernando Luís Dotti[3]

Faculdade de Informática
Pontifícia Universidade Católica do Rio Grande do Sul
Porto Alegre, Brazil

Leila Ribeiro[4]

Instituto de Informática
Universidade Federal do Rio Grande do Sul
Porto Alegre, Brazil

Abstract

Object-Based Graph Grammars (OBGG) is a formal language suitable for the specification of distributed systems. On previous work, a translation from OBGG models to PROMELA (the input language of the SPIN model checker) was defined, enabling the verification of OBGG models using SPIN. This paper builds on these results, where we extend the approach for property specification and define an approach to interpret PROMELA traces as OBGG derivations, generating graphical counter-examples for properties that are not true for an OBGG model.

Keywords: Graph grammars, model checking, visualization of traces.

1 Introduction

In [7], a visual formal specification language suitable for specifying distributed systems was defined. The language is a restriction of graph grammars, called Object-Based Graph Grammars (OBGG). Currently, models in OBGG can be

[1] This work is partially supported by HP Brasil - PUCRS agreement CASCO (32nd TA), PLATUS (CNPq), IQ-Mobile II (CNPq/CNR) and DACHIA (FAPERGS/IB-BMBF) research projects.
[2] Email: osantos@inf.pucrs.br
[3] Email: fldotti@inf.pucrs.br
[4] Email: leila@inf.ufrgs.br

doi:10.1016/j.entcs.2004.02.061

analysed through simulation [5] and verification [6]. Moreover, starting from an OBGG model we can generate code for execution in a real environment, following a straightforward mapping to Java [5]. We also worked on an approach to consider classical failure models for distributed systems, allowing the reasoning about a given model in the presence of a selected failure [8]. By using the methods and tools mentioned above we have defined a framework to assist the development of distributed systems. The innovative aspect of this framework is the use of the same formal specification language (OBGG) as the underlying unifying formalism.

In [6] an approach for verifying OBGG was defined, where OBGG models are translated to PROMELA, the input language of the SPIN model checker. One important aspect of this approach is that properties (to be verified) are specified over OBGG models, instead of over translated PROMELA models. The proposed approach considered only properties over events (that, in case of OBGG, are rule applications). However, complementary aspects of a system could be specified by using properties over states. Besides, counter-examples obtained from SPIN have no corresponding meaning for OBGG models, because they are automatically generated over translated PROMELA models. The main contributions of this paper are: **(i)** to extend the property specification approach to consider states of objects; **(ii)** to define an approach to generate graphical counter-examples for properties that are not true for an OBGG model.

This paper is organised as follows: Section 2 presents an overview of OBGG and SPIN; in Section 3 we briefly explain the translation of OBGG models defined in [6], and present the contributions **(i)** and **(ii)** of this paper; Section 4 present conclusions, related works and future works.

2 Background

In this section we briefly present OBGG and the SPIN model checker.

2.1 Object-Based Graph Grammars

Graphs are a very natural means to explain complex situations on an intuitive level. Graph rules may complementary be used to capture the dynamical aspects of systems. The resulting notion of graph grammars generalises Chomsky grammar from strings to graphs [14]. The basic notions of graph grammars are that the state of a system can be represented by a graph (the system state graph), and from the initial state of the system (the initial graph), the application of rules successively changes the system state.

Here we will use graph grammars as specification formalism for distributed

systems. The construction of such systems will be done componentwise: each component (called entity) is specified as a graph grammar; then, a model of the whole system is constructed by composing instances of the specified components (this model is itself a graph grammar) [2]. Instead of using general graph grammars for the specification of the components, we will use Object-Based Graph Grammars (OBGG) [7]. This choice has two advantages: on the practical side, the specifications are done in an object-based style that is quite familiar to most of the users, and therefore are easy to construct, understand and consequently use as a basis for implementation; on the theoretical side, the restrictions guarantee that the semantics is compositional, reduce the complexity of matching, as well as eases the analysis of the grammar.

OBGG is a restricted form of graph grammars with respect to the kinds of vertices, as well as the configuration of rules to represent object-based concepts. An OBGG consists of a type graph, an initial graph and a set of rules (see Figure 1)[5]. The type graph is actually the description of the (graphical) types that will be used in this grammar (it specifies the kinds of entities, messages, attributes and parameters that are possible – like the structural part of a class description).

The behaviour of an entity when reacting to a message is defined by a (set of) rule(s). Therefore the left-hand side of a rule always specifies the reception of a message by a specific entity. At the right-hand side, that message is consumed and the effect of applying the rule is defined. This effect may be: change of attribute values; creation of objects (instances of entities); and/or generation of new messages. The initial graph specifies the start state of the system.

2.1.1 Example

In this section we model the readers and writers problem using OBGG. The type graph, initial graph and rules for the objects that compose the specification are presented in Figure 1. The readers and writers problem is composed by a resource, and two kinds of processes (readers and writers). Readers and writers processes can execute, read and write operations in the resource, respectively. In order to maintain a consistent state in the resource, each write operation must have exclusive access to the resource. Reader processes can concurrently execute read operations in the resource, provided that no writer process is accessing the resource.

[5] Graphical notation: in Figure 1 (a) rectangles are vertices and numbers inside circles are the names of these vertices (those symbols are used to indicate the type of each vertex in Figures 1 (b), (c), and (d)). The list within a vertex is the vertex attributes. Message symbols that appear in Figure 1 are hyperarcs.

In our modelling of the problem two entities are defined (Figure 1 (a)): *Resource* and *Proc*. The *Resource* entity represents the resource and has two attributes that keep track of the number of readers (*nr*) and writers (*nw*) accessing the resource. The *Proc* entity represents both readers and writers, having an *id* (identification) and a reference to the *Resource* (*res*). The rules for *Resource* objects are shown in Figure 1 (c), and the rules for *Proc* objects are presented in Figure 1 (d). A *Resource* object can accept requests for read operations (rule *ReqRead*) and write operations (rule *ReqWrite*). If a read operation has been accepted, it will eventually finish (rule *FinRead*), sending a confirmation to the *Proc* that sent the request. The same happens to a write operation (rule *FinWrite*). *Proc* objects have a cyclic behaviour, i.e. after receiving confirmation of the operation from the *Resource* (rules *EndRead* and *EndWrite*), a *Proc* object starts a new operation (rules *StartRead* and *StartWrite*). In Figure 1 (b), an initial graph consisting of a *Resource* object, two *Proc* "readers", and two *Proc* "writers" objects is given.

2.2 SPIN

The SPIN model checker [10] is a tool for the verification of concurrent software systems. The input language of SPIN is PROMELA. Property specifications in SPIN are defined using Linear Temporal Logic (LTL). Besides being a model checker, SPIN can be used as a simulator. The simulation feature of SPIN, among other things, is used to simulate counter-examples (traces) of properties that are false for a given model. A counter-example generated by SPIN is composed by statements of the PROMELA model having variables substituted by values of the current state of the model.

PROMELA has a *C-like* syntax and constructs for receiving/sending messages similar to Communication Sequential Processes (CSP). The language is process-based, where processes can exchange information through synchronous and asynchronous message channels and global variables. Non-determinism is modelled using condition and repetition structures.

When specifying properties in SPIN, the user needs to define atomic propositions over the PROMELA model. One possibility for defining atomic propositions is using global variables. This leads to the need to insert attributions to global variables that will be used in verification.

3 Verification of OBGG using SPIN

In this section we present main characteristics of the translation of OBGG models. We describe how properties are modelled, and present an extension of the property specification approach to consider internal state of objects

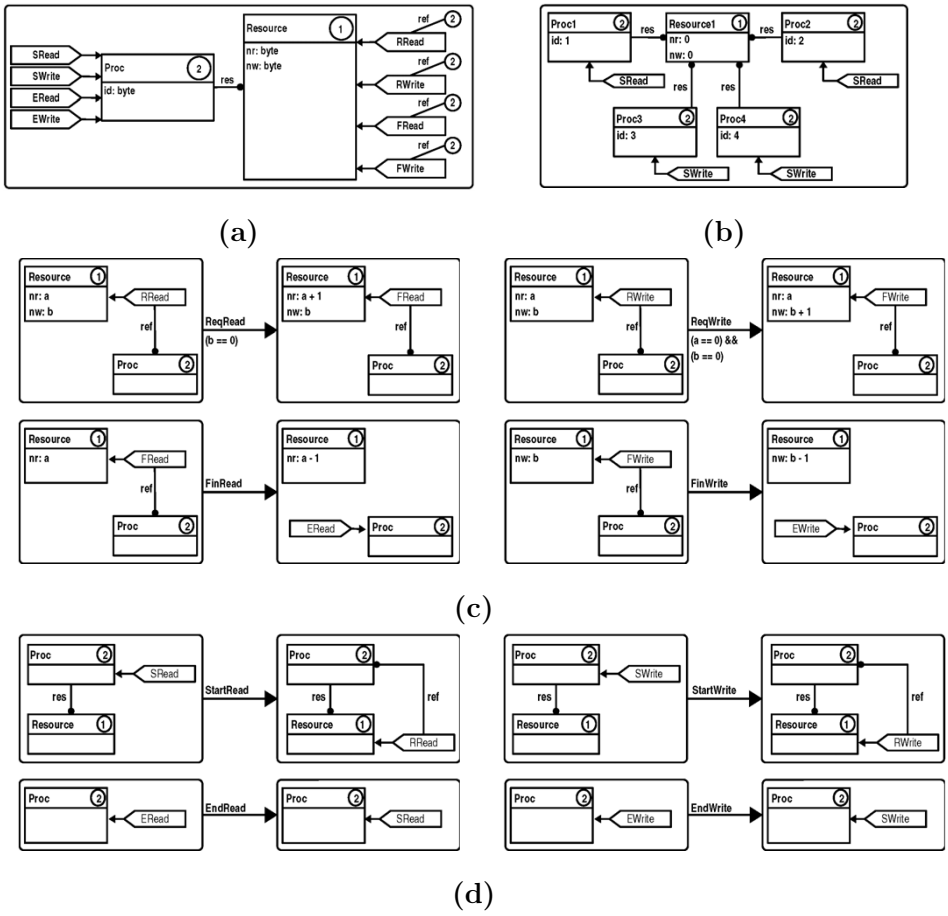

Figure 1. Type graph (a), initial graph (b), and rules (c) and (d).

(Section 3.2). Finally, in Section 3.3 we introduce an approach for generating graphical counter-examples, out of PROMELA traces, that are meaningful to OBGG users.

3.1 Overview of the translation

In the translation defined in [6], OBGG objects are mapped to PROMELA processes (which we call *object processes*). Variables that compose *object processes* are attributes of corresponding objects. For verification purposes, attributes of OBGG objects are restricted to the types supported by PROMELA. OBGG messages are translated to PROMELA messages. The receipt of messages is done through an asynchronous channel, called *object process channel*, also used as reference to the *object process*. Rules for OBGG objects are

mapped to a condition structure inside *object processes*. This structure has in its entries the necessary conditions to trigger the object rules.

Concurrency among objects is naturally preserved by the concurrency between *object processes*. There are two aspects to consider on the use of PROMELA channels: (i) in OBGG the messages to be processed by an object do not preserve ordering - and PROMELA channels do; (ii) in OBGG an unbounded number of messages may be received by an object at each moment - in PROMELA a channel has a maximum number of messages and if the channel is full subsequent writes to the channel will synchronize with eventual reads on the same channel. To deal with (i), each object has an internal channel used to process stored messages in a non-deterministic way. To deal with (ii) we inserted assertions that, just before sending a message, evaluate an expression to determine if the destination channel is not full. Thus, when verifying a model, an error can be generated when an *object process channel* is full, requiring the user to increase the buffer size.

The generic behaviour of an *object process* is as follows: **(i)** wait for new messages in the *object process channel*; **(ii)** once new messages are received, send them to an internal buffer of the *object process*; **(iii)** non-deterministically choose a message from the internal buffer and try to apply a rule to process that message; **(iv. a)** if a message is processed and the *object process channel* is empty, return to **(iii)**; **(iv. b)** if no message is processed or the *object process channel* is not empty, return to **(i)**.

The OBGG initial graph is composed of object instances and messages of the model. The initial graph becomes an *init* process in PROMELA. This *init* process has three stages: **(i)** create *object process channel(s)* for objects appearing in the initial graph; **(ii)** execute *object process(es)* defined in the initial graph; **(iii)** send initial messages using the *object process channel(s)*.

More details about this translation can be found in [6], including a discussion of the semantic compatibility for the generated PROMELA model and the original OBGG model.

3.2 Property specification

Basically, in the literature we can identify two complementary approaches used to specify properties about executions of models, one based on events and another based on states. For using these approaches with OBGG, we can view the application of rules as being events and graphs, obtained from the application of rules, as states. In [6] we provided a way to verify properties based on events for OBGG. This will be presented in more detail in Section 3.2.1. In Section 3.2.2 we present an approach to verify properties considering the internal state of objects.

3.2.1 Properties about events

When specifying properties over translated OBGG models in SPIN, we have to consider that SPIN is an LTL state-based model checker. In order to specify properties based on events, we would have to make available names of the events used in the property specification through global variables. In our approach using events, every PROMELA model generated from the translation has a global variable called *event_RuleName*. The type of this variable is an enumeration of symbolic names which contains the names of OBGG rules that compose the model. Thus, when a rule is applied, the name of the rule is atomically written to the *event_RuleName* global variable, i.e., the name of the applied rule is made visible in the current state of the model.

Using this approach it is possible to write LTL formulas about the occurrence of rules (events) where, for the translated PROMELA model, an event is the change of value of the *event_RuleName* global variable. The idea of specifying properties over events, using SPIN, has been explored in [1]. For instance, we can define an event *ReqRead* (see Figure 1 (c)) as being the change of value of the variable *event_RuleName* from *not ReqRead* to *ReqRead* (where *ReqRead* is the rule being applied). We use the *next* temporal operator (X) to mark the change of value, generating the formula (! *ReqRead* && X *ReqRead*) that represent this event.

As an example of the approach, we can define a LTL formula to specify the mutual exclusion property for the readers and writers problem modelled with OBGG. In order to specify a property using the application of rules as being events, we need to reason about the sequence (order) of rules applications used to represent the property being specified. To represent the mutual exclusion property, we must ensure that read or write operation requests (rules *ReqRead* and *ReqWrite*) are never executed while a write operation is being performed. A write operation is characterized by a request (rule *ReqWrite*) and a confirmation (rule *FinWrite*). Moreover, the mutual exclusion property is defined as "whenever a rule *ReqWrite* (write operation request) is executed and eventually a rule *FinWrite* (write operation confirmation) is applied, none of the rules *ReqWrite* (write operation request) or *ReqRead* (read operation request) is executed in between", leading to the formula ([] (($ReqWrite$ && <> $FinWrite$) $\rightarrow$ X (! ($ReqWrite$ || $ReqRead$) U $FinWrite$))). We verified this formula resulting in a true value from SPIN, where the model used 390 Mb of memory, generated 2.0121e+06 states, and took less than two minutes running in an Intel Xeon 2.2 GHz Processor.

3.2.2 Properties about states

By using the application of rules as events, we can specify properties (in LTL) over OBGG models without having to know the translation. However, we lack the support to express properties about states of objects. When considering the internal state of objects we can, among others, specify properties about specific objects in the model, for instance the impossibility of one specific writer proceed to get the resource (i.e., starvation of that specific writer). In order to address this issue, we extend the approach using events in the sense that users can define events using attributes of objects. Nevertheless, the basis for these events continues to be the application of rules.

For extending the approach we require the user to select attribute(s) she/he wants to make available. For every selected attribute, a global variable is introduced, with syntax *event_EntityName_AttributeName*. Whenever a rule is applied to an object where attribute(s) was(were) selected, the value(s) of the attribute(s) (just after the application of the rule) is(are) copied to the global variable(s).

As an example, we can define a formula to verify that a specific *Proc* "writer" object will never execute a write operation in the resource. We must rely on the utilization of attribute(s) value(s) to identify the desired object in the specification. For the mentioned property, we can use the value of the *id* attribute (which is unique for every *Proc* object in the model) in combination of rules (events) that characterize the execution of a write operation. The execution of a write operation by a *Proc* object is made up by a rule used to start a write operation request (rule *StartWrite*) and another to receive the confirmation that a write operation has succeeded (rule *EndWrite*). Thus, the property is defined as "every time that a rule *StartWrite* (start of write operation request) is executed by a *Proc* object with *id* three, eventually a rule *EndWrite* (the write operation has finished) will be executed by the *Proc* object with *id* value equal to three". We make available the attribute *id* of *Proc* entity, i.e., a global variable *event_Proc_id* is inserted in the model. Then, we define the events: *StartWrite_3* that has as basis rule *StartWrite* and requires the value *id* of the object to be equals to three ($event\_Proc\_id == 3$); and *EndWrite_3* that has as basis rule *EndWrite* and requires, like for event *StartWrite_3*, the value *id* of the object to be equals to three. The LTL formula for this property is ($[]\ (\ StartWrite\_3\ \rightarrow <>\ EndWrite\_3\ )$). We were able to verify this formula resulting in a false value from SPIN. With this, the possibility of starvation by a specific *Proc* object in the model is proved.

The obtained counter-example from SPIN corresponds to a file, where lines of the PROMELA model (translated OBGG model) have variables substituted by values of the current state of the model. In next section we use the obtained

counter-example, for this formula, to illustrate the approach for generating graphical counter-examples that are meaningful for OBGG users.

3.3 Generation of counter-examples

In the literature of distributed systems, one widely accepted graphical form to view the execution of distributed systems has its basis on the exchange of messages between processes. This approach consists on defining a time-line for each process that compose the system. Time increases downwards, by showing the messages (via labelled arcs) being send/received by processes.

Since OBGG has its focus on the specification of distributed systems, the use of a graphical representation similar to the one described has the advantage of being intuitive for users that work with the abstraction of message passing. However, showing only the exchange of messages does not capture another important abstraction of OBGG, the application of rules. In order to consider the application of rules in a graphical execution view of OBGG models, we can add information about rule applications to the time-line of each object that compose the system. This information contains the name of applied rule(s), and is added whenever a rule is executed by an object.

As an example of this approach, the counter-example obtained from SPIN for the property verified in Section 3.2.2 is shown graphically in Figure 2. For each object of the model, a time-line identified by the name of the object is defined. Time increases downwards, where messages are shown via labelled arcs. Rules executed by objects are presented in the right-side of the object's time-line. This execution shows a situation where object *Proc3* issues a write operation in the resource (rule *StartWrite*) that is never executed, because *Proc4* write operations are infinitely often executed in the model. When an infinite cycle is detected in the execution, the phrase *START OF CYCLE* is shown in the graphic. The graphic in Figure 2 is generated automatically by filtering the counter-example obtained from SPIN.

Due to inherent characteristics of the translation from OBGG to PROMELA, in the PROMELA model messages do not have unique *ids* (identifications), an information available in OBGG. Therefore when more than one message with the same type and same attributes are sent to the same object, it is not possible to identify which of the identical messages triggered a specific rule in an object. Although much of the analysis needed can be done with these counter examples, in some cases it may not be possible to analyse the causal relationships among events.

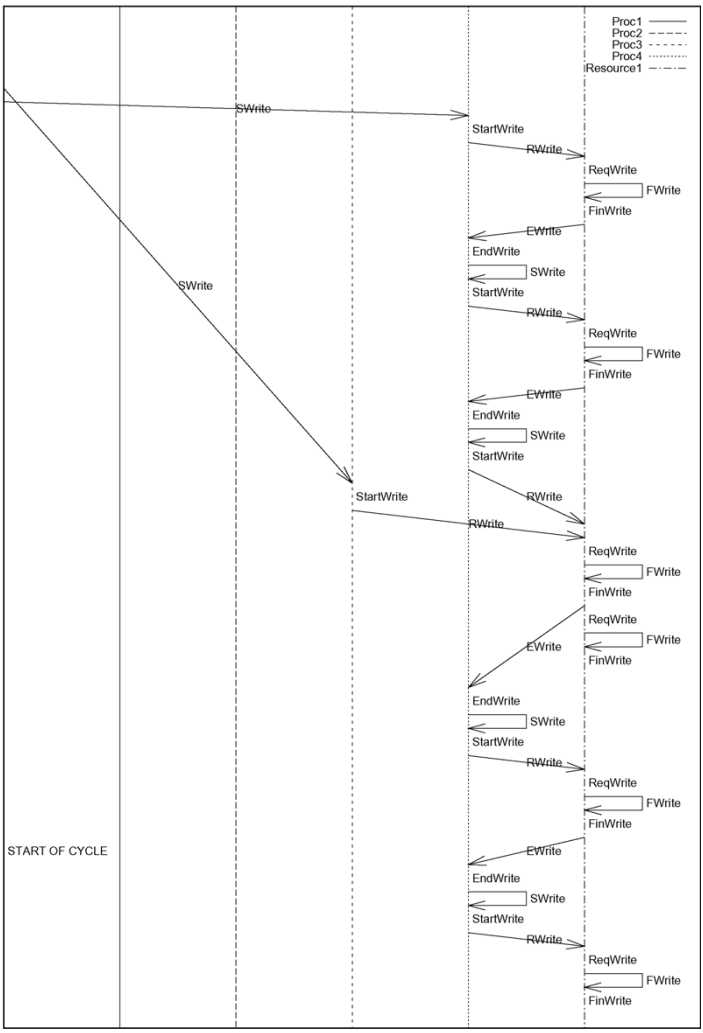

Figure 2. Graphical view of counter-example obtained from SPIN.

4 Final remarks

In this paper we explained how it is possible to specify properties for verification of OBGG models using events (application of rules). Based on that, we defined an extension of the approach to consider the specification of properties using the internal states of objects. The second contribution of this paper is the definition of a graphical layout that is meaningful for OBGG users to view the PROMELA counter-examples.

The translation and integration between formal languages in order to use model checking tools is becoming a common practice, since many times it is easier (and more efficient) to reuse than to build a specific verification tool. We can find in the literature various works, using visual and non-visual languages, focused on the verification of object-based/oriented systems.

In the literature, when focusing on the verification of non-visual languages we found works like [4,3,13,9,16]. More specifically, the works presented in [4,3,9] aim the verification of restricted Java programs, where Java programs are translated to the input language of the SPIN model checker. Despite these works dealing with the verification of Java programs, [13] extends PROMELA (the input language of SPIN) by considering the actors concurrency model, in order to model check object-based distributed systems. In [16] an integration of the formal specification language Object-Z with ASM (*Abstract State Machine*) was introduced, creating the OZ-ASM notation. After a series of translations, it is possible to verify OZ-ASM specifications using the SMV tool.

For the verification of visual languages, we found the following works in the literature [11,12,15]. The work proposed in [11] defines a visual and object-oriented language (called v-Promela) that can be mapped to the SPIN model checker. Property specifications can be defined over v-Promela models, instead of translated PROMELA models, but there is no approach to visualize in terms of v-Promela the results of verifications using SPIN. [12] proposes a tool, called vUML, which tries to make available the automatic verification of UML models. This approach consists in the mapping of UML models to PROMELA. Using the tool, it is possible to verify UML models with respect to deadlocks, livelocks, invalid states, and so on [12]. Moreover, the counter-examples of verifications using SPIN are presented with UML sequence diagrams. In [15], a framework for model checking visual languages is presented. The main idea behind the approach is that visual languages can be modelled as graph transformation system. So, having an approach for verifying general graph transformation systems it is possible to verify visual languages. A manner to specify properties for verification is presented, but we could not find how counter-examples can be generated using the framework.

Differently from most similar approaches considering the translation between languages for model checking, in our work it is possible to specify properties and to view graphical counter-examples in a level of abstraction which is compatible with the language used to specify the model (OBGG), i.e., using the main OBGG abstractions. Moreover, we have a formal proof of the correctness of our translation, something not found on most of the works found in the literature.

Future works consists on defining syntax for the specification of properties and the implementation of a tool for verifying OBGG using the approaches defined so far.

References

[1] Chechik, M. and D. O. Păun, *Events in property patterns*, in: *5th and 6th International SPIN Workshops*, LNCS **1680** (1999), pp. 154–167.

[2] Copstein, B. and L. Ribeiro, *Specifying simulation models using graph grammars*, in: *10th European Simulation Symposium* (1998), pp. 60–64.

[3] Corbett, J. C. et al., *Bandera: extracting finite-state models from java source code*, in: *22nd International Conference on Software Engineering* (2000), pp. 439–448.

[4] Demartini, C., R. Iosif and R. Sisto, *Modeling and validation of java multithreading applications using SPIN*, in: *4th International SPIN Workshop*, France, 1998.

[5] Dotti, F. L., L. M. Duarte, B. Copstein and L. Ribeiro, *Simulation of mobile applications*, in: *2002 Communication Networks and Distributed Systems Modeling and Simulation Conference* (2002), pp. 261–267.

[6] Dotti, F. L., L. Foss, L. Ribeiro and O. M. Santos, *Verification of object-based distributed systems*, in: *6th International Conference on Formal Methods for Open Object-based Distributed Systems*, LNCS **2884** (2003), pp. 261–275.

[7] Dotti, F. L. and L. Ribeiro, *Specification of mobile code systems using graph grammars*, in: *4th International Conference on Formal Methods for Open Object-Based Distributed Systems*, IFIP Conference Proceedings **177** (2000), pp. 45–63.

[8] Dotti, F. L., L. Ribeiro and O. M. Santos, *Specification and analysis of fault behaviours using graph grammars*, in: *2nd International Workshop on Applications of Graph Transformations with Industrial Relevance*, Lecture Notes in Computer Science **3062** (2003), pp. 120–133.

[9] Havelund, K. and T. Pressburger, *Model checking java programs using java pathfinder*, Software Tools for Technology Transfer **2** (2000), pp. 366–381.

[10] Holzmann, G. J., *The model checker SPIN*, IEEE Transactions on Software Engineering **23** (1997), pp. 279–295.

[11] Leue, S. and G. Holzmann, *v-Promela: a visual, object oriented language for SPIN*, in: *2nd International Symposium on Object-Oriented Real-Time Distributed Computing* (1999), pp. 14–23.

[12] Lilius, J. and I. P. Paltor, *vUML: a tool for verifying UML models*, in: *14th International Conference on Automated Software Engineering* (1999), pp. 255–258.

[13] Mo Cho, S. et al., *Applying model checking to concurrent object-oriented software*, in: *4th International Symposium on Autonomous Decentralized Systems* (1999), pp. 380–383.

[14] Rozenberg, G., editor, "Handbook of graph grammars and computing by graph transformations, volume 1: foundations," World Scientific, 1997.

[15] Varró, D., *Towards symbolic analysis of visual modeling languages*, Eletronic Notes in Theoretical Computer Science **72** (2003).

[16] Winter, K. and R. Duke, *Model checking Object-Z using ASM*, in: *3rd International Conference on Integrated Formal Methods*, LNCS **2335** (2002), pp. 165–184.