Educational Aspects of Incremental Model Checking

Ákos DÁVID
Faculty of Information Technology, University of Pannonia
Veszprém, H-8200, Hungary

and

László KOZMA
Department of Software Technology and Methodology
Eötvös Loránd University
Budapest, H-1117, Hungary

ABSTRACT

Model checking has undoubtedly emerged as one of the most effective formal methods for verifying finite, concurrent systems automatically. The most significant problem of model checking is the state space explosion problem occurring in large, complex systems where components can make transitions in parallel. Consequently, extending an existing system with one or more components can lead to an exponential increase in the state space, causing serious problems for model checking tools. The approach of Open Incremental Model Checking (OIMC) is trying to compensate these difficulties by focusing exclusively on the changes of a system instead of rechecking the entire extended system including both the original and the new components. In this paper we study the practical and educational aspects of this new method strongly emphasizing the role of granularity and scalability, meanwhile intending to show also its efficiency. A sample system is used to illustrate the scalability of the extension of a given system.

Keywords: component, model, verification, model checking

1. INTRODUCTION

Model checking has undoubtedly emerged as one of the most effective formal methods for verifying finite, concurrent systems automatically [2]. The most significant problem of model checking is the state space explosion problem occurring in large, complex systems where components can make transitions in parallel. During the past couple of years a considerable progress has been made using the following approaches: symbolic algorithms with Binary Decision Diagrams (BDD), partial order reduction and abstraction.
In [3] the importance of the early design decisions concerning the granularity of components was shown. In [4] a method was given to reduce the state space by using many similar components. In [5] the role of constraints is put into focus to preserve specific properties when a base system is extended with one or more components. These constraints are similar to the contracts introduced by Bertrand Meyer [6], [7].
In the following sections the sample system of an oversimplified airport is given, modified and analyzed with the NuSMV model checker. SMV (Symbolic Model Verifier) is a tool for checking that finite-state systems meet the specifications given in Computation Tree Logic (CTL).
NuSMV is originated from reengineering, reimplementing and extending the code of SMV. The new tool is robust and close to industrial systems standards, making it suitable for modeling life-like systems. The analysis of specifications expressed in Linear Temporal Logic (LTL) is also possible in this version. In NuSMV the specification of the system – usually a state transition machine (STM) – and the constraints imposed on its functioning expressed by CTL formulas are present together.
In Section 2 the sample system of an oversimplified airport is briefly outlined. In Section 2.1 the specification and a model of the airport are given. In 2.2 a separately developed and checked (faulty) airplane component is analyzed whether it can interoperate with the existing airport system. In Section 2.3 an extension of the airport is described and the OIMC algorithm is shown to make an impact on the model checking process. Section 3 presents some concluding remarks focusing on the open questions to be solved.

2. A SAMPLE SYSTEM: AN AIRPORT

A small airport is given with one control tower and a number of airplanes leaving and arriving. An aircraft is on the ground, taking off, flying or landing. The tower and the airplanes are communicating with one another via messages (the airplanes are requesting takeoffs and landings, while the tower is issuing the relevant permits). In the following sections an oversimplified version of the airport with one control tower and two airplanes is modeled and analyzed. On this high level of abstraction the tower is symbolized with a Boolean variable *tower*, and all the communication between the tower and the airplanes is reduced to setting and querying the value of this variable.

The basic version

In the following example the airport system consists of two components (implemented as modules in NuSMV), see Figure 1 for a simplified state diagram of the system. There are two airplanes – entities of the class *plane* – explicitly present while the tower is represented by a Boolean variable *tower*. Each airplane is parameterized with the value of *tower* replacing the entire process of communication with querying and setting the actual value of this variable. The specification is represented by CTL formulas.

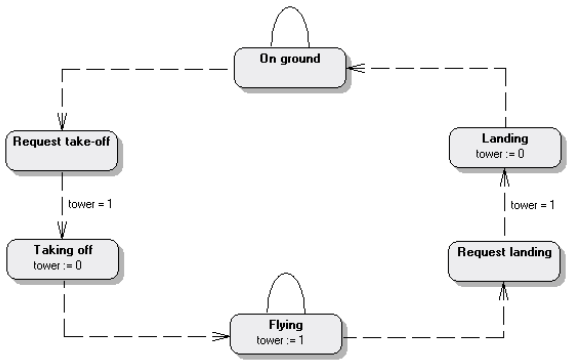

Figure 1: Simplified state diagram of the airport sample system

In the global specification of the airport the formula *SPEC AG(tower = 0 | tower = 1)* should guarantee that the value of *tower* is always 0 or 1. It is quite obvious in this case but in real life situations there might be an occurring distortion of information leading to a possible breakdown of the air traffic control. An example will be given in Section 2.2.
Formula *SPEC AG(tower=1 -> !(plane1.state=taking_off | plane1.state=landing | plane2.state=taking_off | plane2.state=landing))* and formula *SPEC AG(tower=0 -> (plane1.state=taking_off xor plane1.state=landing xor plane2.state=taking_off xor plane2.state=landing))* ensure there are no misunderstood messages during the synchronization of the airplanes so that only one of them can be on the runway of the airport either taking off or landing (in this case when *tower = 0)*.
In the module specification of the airplane the fairness constraints are meant to ignore all those sequences where the tower does not guarantee that each eligible airplane will eventually be chosen for running [1]. This means that a fairness constraint restricts the attention only to fair execution paths. The model checker does not consider path quantifiers not applying to fair paths when there are evaluating specifications. We note that NuSMV supports two types of fairness constraints, namely justice constraints and compassion constraints. A justice constraint consists of a formula *f* which is assumed to be true infinitely often in all the fair paths. A compassion constraint consists of a pair of formulas *(p, q)*; if property *p* is true infinitely often in a fair path, then also formula *q* has to be true infinitely often in the fair path.
Formula *SPEC AG !(state = on_ground & state = flying)* within the module airplane ensures that a nonsense such as a plane in the air and on the ground at the same time is not possible. Formula *SPEC AG (state = flying -> AF (state = landing))* should check that all planes having taken off will eventually land.
The following global specification formulas are situated within the module *airplane* for simply technical reasons. Otherwise, there should be as many of them as the number of airplanes in the system.
Formula *SPEC AG((state = req_takeoff & tower = 1) -> AF(state = taking_off))* guarantees that any plane requesting a takeoff will eventually get the permission if the runway is not used by another plane (when *tower = 1*).

-- specification AG !(state = on_ground & state = flying) IN plane1 is true
-- specification AG (state = flying -> AF state = landing) IN plane1 is true
-- specification AG ((state = req_takeoff & tower = 1) -> AF state = taking_off) IN plane1 is true
-- specification AG ((state = req_takeoff & tower = 0) -> AX !(state = taking_off)) IN plane1 is true
-- specification AG ((state = req_land & tower = 1) -> AF state = landing) IN plane1 is true
-- specification AG ((state = req_land & tower = 0) -> AX !(state = landing)) IN plane1 is true
-- specification AG !(state = on_ground & state = flying) IN plane2 is true
-- specification AG (state = flying -> AF state = landing) IN plane2 is true
-- specification AG ((state = req_takeoff & tower = 1) -> AF state = taking_off) IN plane2 is true
-- specification AG ((state = req_takeoff & tower = 0) -> AX !(state = taking_off)) IN plane2 is true
-- specification AG ((state = req_land & tower = 1) -> AF state = landing) IN plane2 is true
-- specification AG ((state = req_land & tower = 0) -> AX !(state = landing)) IN plane2 is true
-- specification AG (tower = 0 xor tower = 1) is true
-- specification AG (tower = 1 -> !(((plane1.state = taking_off | plane1.state = landing) | plane2.state = taking_off) | plane2.state = landing)) is true
-- specification AG (tower = 0 -> (((plane1.state = taking_off xor plane1.state = landing) xor plane2.state = taking_off) xor plane2.state = landing)) is true

Figure 2: Screenshot of the successful completion of model checking

Formula *SPEC AG((state = req_takeoff & tower = 0) -> AX(!(state = taking_off)))* ensures that none of the airplanes can take off in the next moment if the runway is currently in use. The description of the two formulas concerning landing can be given analogously.
The result of the model checking for this specific case of one tower and two airplanes can be seen in Figure 2. The total number of allocated BDD nodes in this case: 4,242. The example above could also be specified by using LTL formulas. For more flexibility, the extension of LTL with past temporal operators may be used to express certain properties of the airport (e.g. checking whether the doors are locked before takeoff or the wheels are out before landing).

A (faulty) variant of the airplane
Let us take a look at the following variant of the airport system. The occupation of the runway can be interpreted in different ways. For example, the opposite meaning can be associated with the truth values of tower. It can be easily seen that this requires only a slight modification of the module airplane from the previous example described in Section 2.1. The necessary modifications within the model of the airplane can be seen in Figure 3.

...
*init(tower) := **0**;*
...

```
next(tower) :=
case
        (state = req_takeoff) | (state = req_land) : 1;
        (state = taking_off) | (state = landing) : 0;
        1 : tower;
esac;
...
```

Figure 3: Necessary changes to the model of the airplane

The global specification formulas situated in the module *airplane* can be left out as we might consider a COTS (Component Off-the-Shelf) component with no knowledge of the outside world. The total number of allocated nodes in this case is 367. The screenshot of a successful execution of checking the model of this component can be seen in Figure 4.

```
-- specification AG !(state = on_ground & state = flying) IN
plane1 is true
-- specification AG (state = flying -> AF state = landing) IN
plane1 is true
```

Figure 4: Model checking results of a single airplane component

A system that consists of airplanes of this type communicating with a tower using opposite meaning messages compared to the one in the previous example is also functioning properly as it can be proved by using NuSMV, however it is not detailed here for lack of space.
Let us now consider a situation in which the airport described in Section 2.1 is extended with a new airplane of this latter type. The necessary modifications to the specification can bee seen in Figure 5.

```
SPEC AG(tower = 0 | tower = 1)
SPEC AG(tower=1 -> !(plane1.state=taking_off |
        plane1.state=landing |plane2.state=taking_off |
        plane2.state=landing | plane3.state=taking_off |
        plane3.state=landing))
SPEC AG(tower=0 -> (plane1.state=taking_off xor
        plane1.state=landing xor plane2.state=taking_off xor
        plane2.state=landing xor plane3.state=taking_off xor
        plane3.state=landing))
```

Figure 5: Necessary modifications for the verification of the extended system

The first formula is not violated because even with opposite meaning values the interval is still the same. The second and third formulas are violated, and this fact is shown by counterexamples generated with the execution of a standard model checking process.
However, for educational purposes it would be interesting to see how important it is to properly define the sufficient conditions for a minimally safe system. So what happens if the second and third formulas are "forgotten" and left out of the global specification accidentally? The third (faulty) airplane will misinterpret the messages of the tower. In case *tower = 1*, it will wait either on the ground or in the air though the runway is actually empty. In case *tower = 0*, it will not be aware that another plane is using the runway and this mix-up will result in a possible crash. The following screenshot in Figure 6 illustrates that the model checking process will still be successful, though the system is not trustable. This situation is dangerous and unacceptable.

```
-- specification AG !(state = on_ground & state = flying) IN
plane1 is true
-- specification AG (state = flying -> AF state = landing) IN
plane1 is true
-- specification AG ((state = req_takeoff & tower = 1) -> AF
state = taking_off) IN plane1 is true
-- specification AG ((state = req_takeoff & tower = 0) -> AX
!(state = taking_off)) IN plane1 is true
-- specification AG ((state = req_land & tower = 1) -> AF state
= landing) IN plane1 is true
-- specification AG ((state = req_land & tower = 0) -> AX
!(state = landing)) IN plane1 is true
-- specification AG !(state = on_ground & state = flying) IN
plane2 is true
-- specification AG (state = flying -> AF state = landing) IN
plane2 is true
-- specification AG ((state = req_takeoff & tower = 1) -> AF
state = taking_off) IN plane2 is true
-- specification AG ((state = req_takeoff & tower = 0) -> AX
!(state = taking_off)) IN plane2 is true
-- specification AG ((state = req_land & tower = 1) -> AF state
= landing) IN plane2 is true
-- specification AG ((state = req_land & tower = 0) -> AX
!(state = landing)) IN plane2 is true
-- specification AG !(state = on_ground & state = flying) IN
plane3 is true
-- specification AG (state = flying -> AF state = landing) IN
plane3 is true
-- specification AG (tower = 0 xor tower = 1)  is true
```

Figure 6: A faulty system not recognized by model checking because of the insufficient global specification

This phenomenon sheds light on some of the problems of modeling a system as an insufficient specification or the use of bad formulas may lead to unsafe systems undermining the main advantage of formal methods, namely the guarantee that there are no hidden errors left in the model and in the code.

A correct solution improved with OIMC
Now let us consider a situation in which the airport described in Section 2.1 is extended with an airplane originating from the very same class *plane*. This new third plane does not contain the global specification formulas within the module as opposed to the existing aircrafts. After checking the model of this extended system we get the following results shown in Figure 7.

```
-- specification AG !(state = on_ground & state = flying) IN
plane1 is true
-- specification AG (state = flying -> AF state = landing) IN
plane1 is true
-- specification AG ((state = req_takeoff & tower = 1) -> AF
state = taking_off) IN plane1 is true
-- specification AG ((state = req_takeoff & tower = 0) -> AX
!(state = taking_off)) IN plane1 is true
-- specification AG ((state = req_land & tower = 1) -> AF state
= landing) IN plane1 is true
-- specification AG ((state = req_land & tower = 0) -> AX
!(state = landing)) IN plane1 is true
-- specification AG !(state = on_ground & state = flying) IN
plane2 is true
-- specification AG (state = flying -> AF state = landing) IN
plane2 is true
```

-- specification AG ((state = req_takeoff & tower = 1) -> AF state = taking_off) IN plane2 is true
-- specification AG ((state = req_takeoff & tower = 0) -> AX !(state = taking_off)) IN plane2 is true
-- specification AG ((state = req_land & tower = 1) -> AF state = landing) IN plane2 is true
-- specification AG ((state = req_land & tower = 0) -> AX !(state = landing)) IN plane2 is true
-- specification AG !(state = on_ground & state = flying) IN plane3 is true
-- specification AG (state = flying -> AF state = landing) IN plane3 is true
-- specification AG (tower = 0 xor tower = 1) is true
-- specification AG (tower = 1 -> !(((((plane1.state = taking_off | plane1.state = landing) | plane2.state = taking_off) | plane2.state = landing) | plane3.state = taking_off) | plane3.state = landing)) is true
-- specification AG (tower = 0 -> (((((plane1.state = taking_off xor plane1.state = landing) xor plane2.state = taking_off) xor plane2.state = landing) xor plane3.state = taking_off) xor plane3.state = landing)) is true

Figure 7: Result screen of NuSMV showing that the extended system preserves the desired properties

The total number of allocated nodes is 21,075 in case there are three airplanes controlled by the tower. This is significantly greater than the number of nodes in the case of two airplanes.
The scalability of extending the system with more airplanes can be easily guaranteed by forcing the same structure in the subformulas as it can be seen in Figure 8.

SPEC AG(tower=1 -> !(plane1.state=taking_off |
plane1.state=landing | plane2.state=taking_off |
plane2.state=landing | ***plane3.state=taking_off |***
plane3.state=landing*))*
SPEC AG(tower=0 -> (plane1.state=taking_off xor
plane1.state=landing xor plane2.state=taking_off xor
plane2.state=landing xor ***plane3.state=taking_off xor***
plane3.state=landing*))*

Figure 8: Same structure in the subformulas providing scalability

Introducing new airplanes requires a small number of changes in the global specification. These changes are based on the same structure, so it may be done mechanically by the model checker in the future (not yet supported by NuSMV).
The need to reduce the number of states and the recognition of similarity between the components of a system led to a method exploiting the similarities between modules, described in [4]. Unfortunately, the notion of similarity was too restrictive, so a more general solution had to be found.
This way the relatively new approach of OIMC has become the center of attention. The entire theoretical background and the algorithm are beyond the scope of this paper, but an overview can be found in [8]. In the following paragraphs the informal algorithm of OIMC is described exclusively.
In the beginning a property p (in the form of a CTL formula) is assumed to hold in the base component B. Then the extension component E must be checked whether it violates p. By using the incremental model checking algorithm it is sufficient to know whether all the respective pairs of the exit points of B and E preserve the property p. At the end of the execution of the algorithm components B and E consistently preserve the desired property if the truth values of their closures for the property p are the same. There is no need to recheck the entire model of the extended system in addition to the actual verification of the base and the extension.
The closures for the global specification formulas in the base component (one tower and two planes) are the sets of subformulas seen in Figure 9.

Cl_{B1}*=(AG(tower = 0 | tower = 1); (tower = 0 | tower = 1);*
tower=0; tower=1)
Cl_{B2}*=(AG(tower=1 -> !(plane1.state=taking_off |*
plane1.state=landing | plane2.state=taking_off |
plane2.state=landing)); ***tower=1****;*
plane1.state=taking_off; plane1.state=landing;
plane2.state=taking_off; plane2.state=landing)
Cl_{B3}*=(AG(tower=0 -> (plane1.state=taking_off xor*
plane1.state=landing xor plane2.state=taking_off xor
plane2.state=landing)); ***tower=0****;*
plane1.state=taking_off; plane1.state=landing;
plane2.state=taking_off; plane2.state=landing)

Figure 9: The closure of the global specification formulas of the base component

It is quite obvious that in the case of a faulty airplane $Cl_{B1} = Cl_{E1}$ as the new module will use the same values of 0 and 1, meaning that interoperation between the base and the extension is possible. However, the respective pairs of Cl_{B2}, Cl_{E2} and Cl_{B3}, Cl_{E3} will not be equal since the highlighted atomic formulas will have their negated counterparts in their respective pairs.
On the other hand, by extending the airport with an aircraft of the original class plane will result in the same closures for the properties above at the exit points in the extension, making it possible for them to interoperate with each other.
The cost of this type of model checking will be significantly lower than the standard process as the number of allocated nodes in this case is approximately 4,242 for the base and 367 for the extension, resulting in a roughly 78% reduction in the state space. But the cost of determining the exit points, then generating and comparing the respective closures of the base and the extension should also be calculated with some measurement techniques.
The real situation is quite ambiguous as the results of the OIMC algorithm are promising, but the algorithm has not been implemented yet so the entire process needs to be done manually. In practice, it is currently no match for standard model checking techniques.
From an educational point of view it could be a cornerstone in shifting the attention of students from coding to the incremental architecture of systems by using components.

3. CONCLUSIONS

OIMC is a relatively new approach emphasizing the changes to a system rather than model checking the entire system. OIMC was brought to life by the need to reduce the state space significantly in order to make this new generation of model checking techniques applicable in large and complex real life applications.
Currently there are three major problems with the usage of the OIMC algorithm. First, it is not able to handle circular dependency between the interface states of the base and the extension. Secondly, it supports only the CTL temporal logic language.

Thirdly, to our best knowledge, the algorithm has not been implemented in an open-source, close-to-standards model checking tool yet. We are addressing the first problem with a greatest fixed point approach. A possible solution to the second problem is currently underway, the algorithm is modified by introducing new rules to handle LTL formulas and also Past Tense operators. We are also working on the third problem as we firmly believe these results can help in the formal verification of larger and more complex software systems, not only from a practical but also from an educational aspect.

4. REFERENCES

[1] Cavada, R., Cimatti, A., Olivetti, E., Keighren, G., Pistore, M., Roveri, M., Semprini, S., Tchaltsev, A.: **NuSMV 2.3 User Manual**, Retrieved on 06/06/07 from http://nusmv.irst.itc.it/NuSMV/userman/v23/nusmv.pdf

[2] Clarke, E., Grumberg, O., Peled, D.: **Model Checking**, MIT Press, 2000.

[3] Dávid, Á., Kozma, L., Pozsgai, T.: **On the granularity of components**, Proceedings of the 7th International Conference on Applied Informatics, Eger, Hungary, January 28-31, 2007, Vol. 2, pp. 219-228.

[4] Dávid, Á., Kozma, L., Pozsgai, T.: **"On the model checking of a system consisting of many similar components"**, Annales Univ. Sci. Budapest., Sect. Comp. 28, pp. 183-195, 2008.

[5] Dávid, Á., Pozsgai, T., Kozma, L.: **"Extending a system with verified components"** presented at CSCS'06, June 27-30, 2006, Szeged, Hungary (full paper accepted for publication by Polytechnica Periodica)

[6] Gross, H. G.: **Component-Based Software Testing with UML**, Springer, 2005.

[7] Meyer, B.: **Object-Oriented Software Construction Second Edition**, Prentice Hall, 1997.