Annales Univ. Sci. Budapest., Sect. Comp. 28 (2008) 183-195

ON THE MODEL CHECKING OF A SYSTEM CONSISTING OF MANY SIMILAR COMPONENTS

Á. Dávid (Veszprém, Hungary)[1]
L. Kozma (Budapest, Hungary)
T. Pozsgai (Veszprém, Hungary)

Dedicated to Professor Imre Kátai on his 70th birthday

Abstract. Model checking can be an efficient way of verifying large, complex software systems. Symbolic model checking tools like NuSMV (an extension of Symbolic Model Verifier) usually expect the model of a system to be specified by Computational Tree Logic formulas. In the component-based world, however, it may be reasonable to use Linear Temporal Logic formulas with special attention to the past tense operators to describe the functioning of a system built of isolated components by means of contracts. In this way not only the portability of specific components is better supported, but also the overall functioning of the system may be easier to understand as most of the constraints are present locally at the related component's level.

In this paper we illustrate how the analysis of the structure of behavioral constraints (usually in the form of LTL formulas) in the local specification of similar components may result in the reduction of the state space of the system model, making the process of model checking considerably faster.

1. Introduction

Our dependence on the correct functioning of computer systems in our daily life becomes more and more evident. Considering the automatic control of heating systems or the safety systems of cars, faults – not even the subtle

[1]This research was supported by GVOP-3.2.2-2004-07-005/3.0.

ones – are not tolerated. Only formal verification methods are able to guarantee that a software system is error-free from the design to the deployment of the actual code.

Of the most widespread formal methods model checking has already proved its effectiveness in handling real-world problems. However, in one of today's favorite paradigms, component-based software development (CBSD) it encounters the problem of separately developed components, components-off-the-shelf (COTS) used in a different environment from the original one [6, 1].

It may sound reasonable to take into consideration of building the information on their correctness into the design and the code of components [2]. The basis of this concept is the design-by-contract (DBC) introduced by Bertrand Meyer [4] and later extended by Reussner [5].

The advantages of structural similarities among components can also be exploited to reduce the number of formulas in the local specification of each component. Please note that in this paper the notion of similarity is restricted to the various entities originating from the very same class. It is desirable but also beyond the scope of this paper to extend the notion of similarity among components to increase the efficiency of this approach in real-world situations.

The idea is that components may be considered to be the implementations of entities originating from one abstract class, in which specification formulas can be separated according to whether they refer only to states and variables within a given component. As these formulas have no effect on any other component of the system, they and their truth values are stored temporarily and consequently need to be checked only once when encountering one or more similar modules. In other words, model checking is shifted to a higher, "class" level. In this case the relevant formulas can be ignored, thus reducing the time and memory needs of model checking.

2. Basic concepts of model checking

Model checking is an algorithmic way of verifying software systems formally. This can be done automatically by verifying whether a specific model meets the expectations of a formal specification. The specification is usually available as a set of temporal logic formulas describing given properties of a system.

The model is preferably given as a source code description in a special-purpose language. Such a program corresponds to a finite state machine (FSM), usually a directed graph consisting of nodes and edges. A set of atomic propositions is associated with each node. The nodes represent states of a system, the edges represent possible transitions which may alter the state, while the atomic propositions represent the basic properties that hold at a point of execution. In some cases infinite systems may also be verified using model checking in combination with various abstraction and induction principles.

The main challenge in applying model checking in real-world problems is dealing with the state space explosion problem. This problem occurs in systems with many components that can make transitions in parallel. During the past few years a considerable progress has been made using the following approaches.

- **Symbolic algorithms** do not actually build the graph for the FSM, they would rather represent the graph implicitly using a formula in propositional logic. Much of the increase has been due to the use of binary decision diagrams (BDD), a data structure for representing Boolean functions recommended by Ken McMillan [3]. Recently, Satisfiability Problem (SAT) solvers have been used to perform the graph search.
- **Partial order reduction** can be used to reduce the number of independent interleavings of concurrent processes that must be considered. The basic idea is that if two interleaving sequences are not distinguished by a given specification then it is sufficient to analyze only one of them.
- **Abstraction** uses a different approach by removing the unnecessary details of a system, thus simplifying the verification process. The simplified system usually does not satisfy exactly the same properties as the original one, so that a process of refinement may be necessary.
- **Open Incremental Model Checking (OIMC)** is a relatively new approach focusing on the changes of a system instead of re-checking the entire system. An overview of this method and the algorithm used can be found in [7, 8].

Model checking tools use their own specification languages, but most of them support the temporal logic languages LTL and CTL. In Sections 2 and 2 we give an overview of the syntax and semantics of LTL with the extension of past temporal operators.

2.1. Syntax of LTL

The logic LTL is a linear temporal logic, meaning that the formulas are interpreted over infinite sequences of states. A minimal syntax for LTL formulas can be given in the following way. Given the non-empty set of atomic propositions (AP), an LTL formula is:

- $true, false$ or p where $p \in AP$,
- $\neg f_1$, $f_1 \vee f_2$, Xf_1 (read "next time") or $f_1\ U\ f_2$ (read "until"), where f_1 and f_2 are LTL formulas,

It is often necessary for describing real-world problems to use a non-minimal version of the LTL syntax. Given the set AP, an LTL formula is:

- $true, false$ or p, where $p \in AP$,
- $\neg f_1$, $f_1 \vee f_2$, $f_1 \wedge f_2$, Xf_1, $f_1 U f_2$ or $f_1 R f_2$ (read "f_1 releases f_2", meaning f_2 is either always true, or f_2 is true until $f_1 \wedge f_2$ is true), where f_1 and f_2 are LTL formulas,
- **Past temporal operators:** Yf_1 (read "previously"), Hf_1 (read "historically" or "always in the past"), Of_1 (read "once") or $f_1\ S\ f_2$ (read "f_1 since f_2") where f_1 and f_2 are LTL formulas.

Some often used temporal logic shorthands are:

- $Ff_1 = true\ U\ f_1$ (read "finally f_1"),
- $Gf_1 = \neg F \neg f_1$ (read "globally f_1").

2.2. Semantics of LTL

A path π is an infinite sequence of states $\pi = s_0, s_1, s_2, \ldots$, where $(s_i, s_{i+1}) \in$ $\in R$ (an $S \times S$ transition relation) holds for all $i \geq 0$. $(\pi, s_i) \models f$ denotes that an LTL formula f holds in a world (π, s_i), in other words (π, s_i) is a model of the formula f. Model checking refers to the process of checking whether the behaviors of the system are models of the specification formulas. The relation $(\pi, s_i) \models f$ is defined inductively as follows:

- $(\pi, s_i) \models true$,
- $(\pi, s_i) \not\models false$,
- $(\pi, s_i) \models p$ iff $p \in L(s_i)$ where $p \in AP$ and L is a function labeling each s_i with the atomic propositions holding,
- $(\pi, s_i) \models \neg f_1$ iff not $(\pi, s_i) \models f_1$,
- $(\pi, s_i) \models f_1 \vee f_2$ iff $(\pi, s_i) \models f_1$ or $(\pi, s_i) \models f_2$,
- $(\pi, s_i) \models f_1 \wedge f_2$ iff $(\pi, s_i) \models f_1$ and $(\pi, s_i) \models f_2$,

- $(\pi, s_i) \models X f_1$ iff $(\pi, s_{i+1}) \models f_1$,
- $(\pi, s_i) \models f_1 \ U \ f_2$ iff for all $j \geq i$ $(\pi, s_j) \models f_1$ or there is $j \geq i$ such that $(\pi, s_j) \models f_2$ and for all $i \leq k < j$, $(\pi, s_k) \models f_1$,
- $(\pi, s_i) \models f_1 \ R \ f_2$ iff for every $j \geq i$ $(\pi, s_j) \models f_2$ or there is $j \geq i$ that $(\pi, s_j) \models f_1 \wedge f_2$ and for every $i \leq k < j$ $(\pi, s_j) \models f_2$.

If we want, we can also express the semantics of $F f_1$ and $G f_1$ directly:

- $(\pi, s_i) \models F f_1$ iff there is $j \geq i$, that $(\pi, s_j) \models f_1$,
- $(\pi, s_i) \models G f_1$ iff for all $j \geq i$, $(\pi, s_j) \models f_1$.

The semantics of the past temporal operators can also be expressed:

- $(\pi, s_i) \models Y f_1$ iff $(\pi, s_{i-1}) \models f_1$, where $i > 0$;
- $(\pi, s_i) \models H f_1$ iff for all $0 \leq j \leq i$ $(\pi, s_j) \models f_1$;
- $(\pi, s_i) \models O f_1$ iff there is $0 \leq j \leq i$ such that $(\pi, s_j) \models f_1$;
- $(\pi, s_i) \models f_1 \ S \ f_2$ iff for all $0 \leq j \leq i$ $(\pi, s_j) \models f_1$ or there is $0 \leq j \leq i$ such that $(\pi, s_j) \models f_2$ and for all $j < k \leq i$, $(\pi, s_k) \models f_1$.

3. A sample system: Airport

Given a small airport with one control tower and a number of airplanes leaving and arriving. An aircraft is either on the ground, taking off, flying or landing. The tower and the airplanes are communicating with one another via messages (the airplanes requesting takeoffs and landings, while the tower issuing the relevant permits). In Chapter 3 an oversimplified version of the airport with one control tower and one airplane is modeled and analyzed. On this high level of abstraction the tower is symbolized as a boolean variable *tower*, and all the communication between the tower and the airplane is reduced to setting and querying the value of this variable. The simplified functioning of the control tower can be seen in Figure 1 while the state transition model (STM) of the airplane can be seen in Figure 2.

In Chapter 3 the functioning of the airport is extended to handle more airplanes by one control tower. Direct communication between the airplanes is not allowed (they should not be aware of one another) in this case because it is

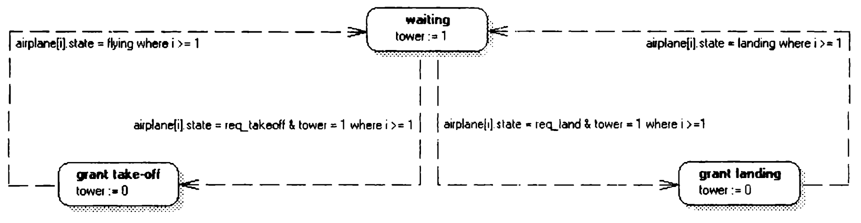

waiting
tower := 1
airplane[i].state = flying where i >= 1
airplane[i].state = landing where i >= 1
airplane[i].state = req_takeoff & tower = 1 where i >= 1
airplane[i].state = req_land & tower = 1 where i >=1
grant take-off
tower := 0
grant landing
tower := 0

Figure 1: A simplified state diagram of the tower

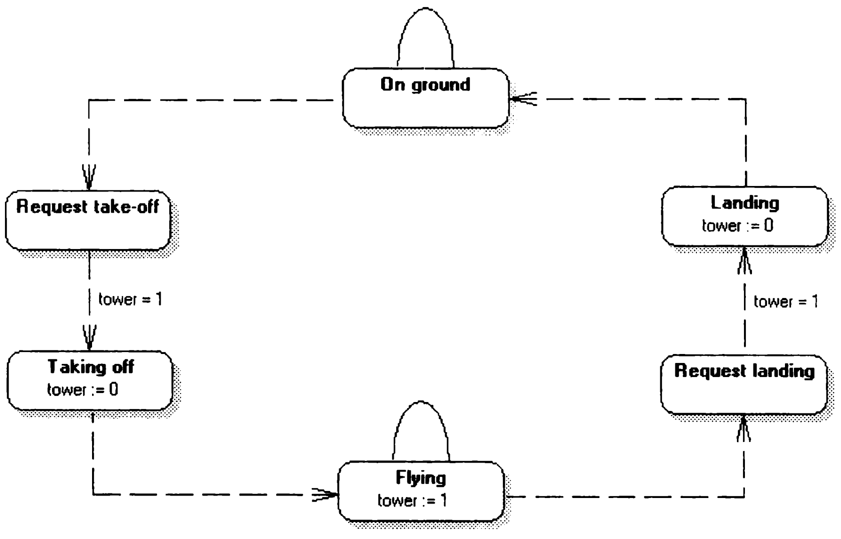


Figure 2: A simplified state diagram of an airplane

the responsibility of the control tower to synchronize the airplanes so that only one of them can be on the runway of the airport either taking off or landing.

3.1. Airport model (with one tower and one airplane)

In the following code excerpt the airport system consists of two components (implemented as modules in NuSMV). The airplane is explicitly present while the tower is represented by a boolean variable *tower*. The 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 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).

The following model of the airplane implemented as module ***plane*** is a one-to-one mapping of the UML state transition diagram shown in Figure 2.

```
MODULE main
VAR
  tower : boolean;
  plane1 : process plane(tower);
ASSIGN
  init(tower) := 1;

LTLSPEC G(tower=0 xor tower=1)

MODULE plane(tower)
VAR
  state : {on_ground,req_takeoff,taking_off,flying,req_land,landing};
ASSIGN
  init(state) := on_ground;
  next(state) :=
    case
      state = on_ground : {on_ground,req_takeoff};
      (state = req_takeoff) & (tower) : taking_off;
      state = taking_off : flying;
      -- extra condition for wheels in
      state = flying : {flying,req_land};
      (state = req_land) & (tower) : landing;
      -- extra condition for wheels out
      state = landing : on_ground;
      1 : state;
    esac;
  next(tower) :=
    case
```

```
      (state = req_takeoff) xor (state = req_land) : 0;
      (state = taking_off) xor (state = landing) : 1;
      1 : tower;
    esac;

FAIRNESS !(state = req_takeoff)
FAIRNESS !(state = flying)
FAIRNESS !(state = req_land)

LTLSPEC G!(state=on_ground & state=flying)
LTLSPEC G(state=flying -> F(state=landing))
LTLSPEC G(state=flying -> O(state=taking_off))
LTLSPEC G(state=taking_off -> O(state=req_takeoff))
LTLSPEC G(state=req_takeoff -> O(state=on_ground))
LTLSPEC G(state=landing -> O(state=req_land))
LTLSPEC G(state=req_land -> O(state=flying))
```

This is a part of the global specification placed within a local module because otherwise the services provided by NuSMV would require the modification of these formulas each time a new entity is introduced into the system. For this reason they were built into the module *plane*.

```
LTLSPEC G ((state = req_takeoff & tower = 1) -> F (state = taking_off))
LTLSPEC G((state = req_takeoff & tower = 0) -> X(!(state = taking_off)))
LTLSPEC G ((state = req_land & tower = 1) -> F (state = landing))
LTLSPEC G((state = req_land & tower = 0) -> X(!(state = landing)))
LTLSPEC G (tower = 1 -> !(state = taking_off | state = landing))

FAIRNESS
  running
```

There are also comments in the module of the airplane referring to some extra conditions that may be applied to make the model more life-like.

3.2. Airport model (with one tower and more airplanes)

In this example the airport is extended to handle more airplanes at a time. The behavioral specification of the airplane is basically the same but the global specification needs to be extended with the following formulas to verify the correct interpretation of messages (*tower* = 1 means the tower is free and the runway is clear, not the other way around) and the proper scheduling of the airplanes in the area:

```
LTLSPEC G(tower=1 -> !(plane1.state=taking_off |
                   plane1.state=landing | plane2.state=taking_off |
                    plane2.state=landing))
LTLSPEC G(tower=0 -> (plane1.state=taking_off xor
                    plane1.state=landing xor plane2.state=taking_off xor
                    plane2.state=landing))
```

In the modular specification the formulas are separated according to whether they refer to variables and states of other modules (the main module is also included) or not. In the first case re-checking of the formulas is unavoidable as there is explicit dependency between components that may be affected by other parts of the system. However, in the latter case there is no need to re-check the formulas that are not crossing the "borders" of the modules. The relevant formulas to be ignored are the following:

```
LTLSPEC G !(state = on_ground & state = flying)
LTLSPEC G (state = flying -> F (state = landing))
LTLSPEC G (state = flying -> O (state = taking_off))
LTLSPEC G (state = taking_off -> O (state = on_ground))
LTLSPEC G (state = req_takeoff -> O (state = on_ground))
LTLSPEC G (state = landing -> O (state = req_land))
```

In this way the model checking of similar components becomes easier with a significant reduction on the state space of the model.

A challenging, open question is whether it is possible to further reduce the resource needs of model checking by adapting and applying Open Incremental Model Checking to handle incremental models with LTL specifications using past tense operators. Studying the possibility of giving a modified algorithm and implementing it in NuSMV is currently underway.

Below is the model including the specification for the extended version of the airport described earlier in this section.

```
MODULE main
VAR
  tower : boolean;
  plane1 : process plane(tower);
  plane2 : process plane(tower);
ASSIGN
  init(tower) := 1;

-- from the original specification
LTLSPEC G(tower = 0 xor tower = 1)

-- begin of the extended formulas
LTLSPEC G(tower=1 -> !(plane1.state=taking_off |
                    plane1.state=landing | plane2.state=taking_off |
                    plane2.state=landing))
```

```
LTLSPEC G(tower=0 -> (plane1.state=taking_off xor
                      plane1.state=landing xor plane2.state=taking_off xor
                      plane2.state=landing))
-- end of the extended formulas

MODULE plane(tower)
```

The model describing the functioning of the airplane is exactly the same as in the previous example, so it is not detailed here. Fairness constraints are also unchanged. The following formulas can be removed from each module other than the first one representing the abstract class.

```
 LTLSPEC G !(state = on_ground & state =
flying) LTLSPEC G (state = flying -> F (state = landing)) LTLSPEC G
(state = flying -> O (state = taking_off)) LTLSPEC G (state =
taking_off -> O (state = on_ground)) LTLSPEC G (state = req_takeoff
-> O (state = on_ground)) LTLSPEC G (state = landing -> O (state =
req_land))
```

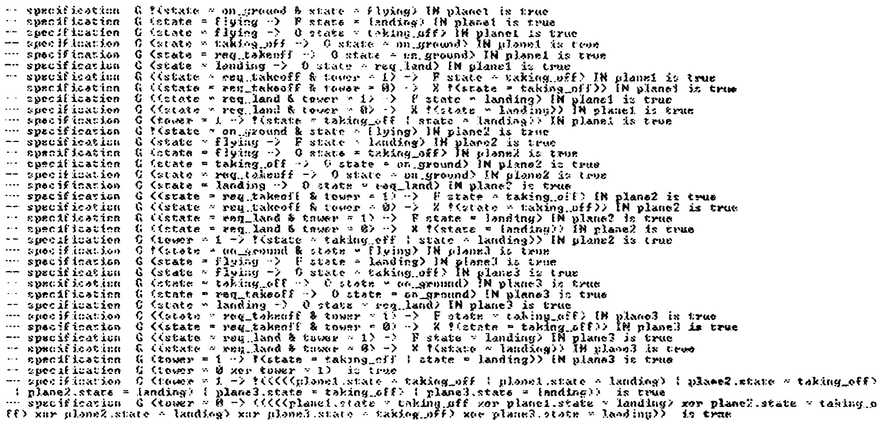
The following part is also unchanged with respect to the previous example, but these formulas cannot be trusted without being re-checked each time the system is extended with a new entity as they contain references to the actual state of the tower (variable) making them dependable on that.

```
LTLSPEC G ((state = req_takeoff & tower = 1) -> F (state = taking_off))
LTLSPEC G((state = req_takeoff & tower = 0) -> X(!(state = taking_off)))
LTLSPEC G ((state = req_land & tower = 1) -> F (state = landing))
LTLSPEC G((state = req_land & tower = 0) -> X(!(state = landing)))
LTLSPEC G (tower = 1 -> !(state = taking_off | state = landing))

FAIRNESS
  running
```

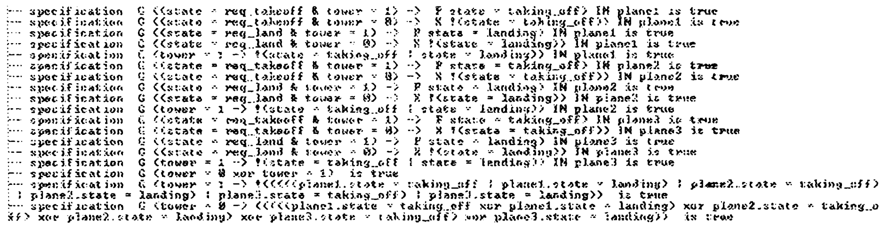
Running a model check without removing the formulas identified in the previous section results in the screen illustrated by Figure 3. The total numbers of allocated nodes in the case of 2, 3 and 4 airplanes are 49,666; 250,501; 685,005 respectively.

Applying the reduction rules described earlier to force the model checker to ignore the relevant formulas results in the screen shown by Figure 4. The number of formulas checked, the size of the state space, just like the time and memory needs of model checking are seemingly down. The total numbers of allocated nodes in the case of 2, 3 and 4 airplanes are 32,801 (34% reduction); 185,284 (26% reduction); 530,401 (23% reduction), respectively. The pattern is too small to provide sufficient base for further conclusions, but the fact of the reduction in the state space is demonstrated well.



```
-- specification  G !(state = on_ground & state = flying) IN plane1 is true
-- specification  G (state = flying -> F state = landing) IN plane1 is true
-- specification  G (state = flying -> O state = taking_off) IN plane1 is true
-- specification  G (state = taking_off -> O state = on_ground) IN plane1 is true
-- specification  G (state = req_takeoff -> O state = on_ground) IN plane1 is true
-- specification  G (state = landing -> O state = req_land) IN plane1 is true
-- specification  G ((state = req_takeoff & tower = 1) -> F state = taking_off) IN plane1 is true
   specification  G ((state = req_takeoff & tower = 0) -> X !(state = taking_off)) IN plane1 is true
-- specification  G ((state = req_land & tower = 1) -> F state = landing) IN plane1 is true
-- specification  G ((state = req_land & tower = 0) -> X !(state = landing)) IN plane1 is true
-- specification  G (tower = 1 -> !(state = taking_off | state = landing)) IN plane1 is true
-- specification  G !(state = on_ground & state = flying) IN plane2 is true
-- specification  G (state = flying -> F state = landing) IN plane2 is true
-- specification  G (state = flying -> O state = taking_off) IN plane2 is true
-- specification  G (state = taking_off -> O state = on_ground) IN plane2 is true
-- specification  G (state = req_takeoff -> O state = on_ground) IN plane2 is true
-- specification  G (state = landing -> O state = req_land) IN plane2 is true
-- specification  G ((state = req_takeoff & tower = 1) -> F state = taking_off) IN plane2 is true
-- specification  G ((state = req_takeoff & tower = 0) -> X !(state = taking_off)) IN plane2 is true
-- specification  G ((state = req_land & tower = 1) -> F state = landing) IN plane2 is true
-- specification  G ((state = req_land & tower = 0) -> X !(state = landing)) IN plane2 is true
-- specification  G (tower = 1 -> !(state = taking_off | state = landing)) IN plane2 is true
-- specification  G !(state = on_ground & state = flying) IN plane3 is true
-- specification  G (state = flying -> F state = landing) IN plane3 is true
-- specification  G (state = flying -> O state = taking_off) IN plane3 is true
-- specification  G (state = taking_off -> O state = on_ground) IN plane3 is true
-- specification  G (state = req_takeoff -> O state = on_ground) IN plane3 is true
-- specification  G (state = landing -> O state = req_land) IN plane3 is true
-- specification  G ((state = req_takeoff & tower = 1) -> F state = taking_off) IN plane3 is true
-- specification  G ((state = req_takeoff & tower = 0) -> X !(state = taking_off)) IN plane3 is true
-- specification  G ((state = req_land & tower = 1) -> F state = landing) IN plane3 is true
-- specification  G ((state = req_land & tower = 0) -> X !(state = landing)) IN plane3 is true
-- specification  G (tower = 1 -> !(state = taking_off | state = landing)) IN plane3 is true
-- specification  G (tower = 0 xor tower = 1)  is true
-- specification  G (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  G (tower = 0 -> (((((plane1.state = taking_off xor plane1.state = landing) xor plane2.state = taking_o
ff) xor plane2.state = landing) xor plane3.state = taking_off) xor plane3.state = landing))  is true
```

Figure 3: NuSMV result screen without removing some of the formulas



```
-- specification  G ((state = req_takeoff & tower = 1) -> F state = taking_off) IN plane1 is true
-- specification  G ((state = req_takeoff & tower = 0) -> X !(state = taking_off)) IN plane1 is true
-- specification  G ((state = req_land & tower = 1) -> F state = landing) IN plane1 is true
-- specification  G ((state = req_land & tower = 0) -> X !(state = landing)) IN plane1 is true
-- specification  G (tower = 1 -> !(state = taking_off | state = landing)) IN plane1 is true
-- specification  G ((state = req_takeoff & tower = 1) -> F state = taking_off) IN plane2 is true
-- specification  G ((state = req_takeoff & tower = 0) -> X !(state = taking_off)) IN plane2 is true
-- specification  G ((state = req_land & tower = 1) -> F state = landing) IN plane2 is true
-- specification  G ((state = req_land & tower = 0) -> X !(state = landing)) IN plane2 is true
-- specification  G (tower = 1 -> !(state = taking_off | state = landing)) IN plane2 is true
-- specification  G ((state = req_takeoff & tower = 1) -> F state = taking_off) IN plane3 is true
-- specification  G ((state = req_takeoff & tower = 0) -> X !(state = taking_off)) IN plane3 is true
-- specification  G ((state = req_land & tower = 1) -> F state = landing) IN plane3 is true
-- specification  G ((state = req_land & tower = 0) -> X !(state = landing)) IN plane3 is true
-- specification  G (tower = 1 -> !(state = taking_off | state = landing)) IN plane3 is true
-- specification  G (tower = 0 xor tower = 1)  is true
-- specification  G (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  G (tower = 0 -> (((((plane1.state = taking_off xor plane1.state = landing) xor plane2.state = taking_o
ff) xor plane2.state = landing) xor plane3.state = taking_off) xor plane3.state = landing))  is true
```

Figure 4: NuSMV result screen after removing some of the formulas

The organization of the components guarantees the scalability of extending the system with more airplanes without any difficulties. 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 (not yet supported by NuSMV).

```
LTLSPEC G(tower=1 -> !(plane1.state=taking_off |
                       plane1.state=landing | plane2.state=taking_off |
                       plane2.state=landing | plane3.state=taking_off |
                       plane3.state=landing | ... | ... ))
LTLSPEC G(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 xor ... xor ... ))
```

4. Conclusions

Building a system from components is supposed to ease the burden on software architects by making possible to reuse already existing parts of software, but is also full of challenges because the deployment environments and purposes can severely differ from the original ones.

Model checking is a formal method to verify whether the model of a system is correct to a given specification. The main challenge concerning the usability of model checking in the business world is to handle state space explosion. Building a system from many similar components can help to cope with this problem by making unnecessary to check a subset of formulas in the local specification of components as we have shown in a life-like example.

References

[1] **Bertolino A. and Polini A.,** A framework for component deployment testing, *Proc. 25th Int. Conf. on Software Engineering, May 2003,* 221-231.

[2] **Beugnard A., Jézéquel J.M., Plouzeau N. and Watkins D.,** Making components contract aware, *IEEE Software,* **32** (7) (1999), 38-44.

[3] **McMillan K.L.,** *Symbolic model checking: An approach to the state explosion problem,* Kluwer, 1993.

[4] **Meyer B.,** *Object-oriented software construction,* Prentice Hall, 1997.

[5] **Reussner R.H.,** The use of parameterised contracts for architecting systems with software components, *6th Int. Workshop on Component-Oriented Programming, Budapest, Hungary, 2001.*

[6] **Sparling M.,** Lessons learned through six years of component-based development, *Communications of ACM,* **43** (2000), 47-53.

[7] **Thang N.T. and Katayama T.,** Open incremental model checking (extended abstract), *Proc. of ACM SIGSOFT Symposium on the Foundations of Software Engineering, Newport Beach, California, USA, Oct 31-Nov 1, 2004,* 134-137.

[8] **Thang N.T. and Katayama T.,** Specification and verification of intercomponent constraints in CTL, *ACM SIGSOFT Software Engineering Notes,* **31** (2) (2005), 1-8.

Á. Dávid and T. Pozsgai
Department of Mathematics and Computing
University of Pannonia
Egyetem u. 10.
H-8200 Veszprém, Hungary
davida@almos.uni-pannon.hu
pozsgai@szt.vein.hu

L. Kozma
Department of Software Technology and Methodology
Eötvös Loránd University
Pázmány Péter sét. 1/C
H-1117 Budapest, Hungary
kozma@ludens.elte.hu