VERIFICATION OF A LARGE DISCRETE SYSTEM USING ALGEBRAIC METHODS

Johan Gunnarsson Jonas Plantin Roger Germundsson
Department of Electrical Engineering
Linköping University
S-581 83 Linköping, Sweden
Email: {johan,jonas,roger}@isy.liu.se
WWW: http://www.control.isy.liu.se

WODES96

Keywords
DEDS, Verification, Algebra, BDD, Application, Large

Abstract

Symbolic algebraic modeling and analysis techniques for DEDS are applied to the landing gear subsystem in the new Swedish fighter aircraft, JAS 39 Gripen. Our methods are based on polynomials over finite fields. Polynomials are used to represent the basic dynamic equations for the processes (controller and plant) as well as static properties of these. Temporal algebra (or temporal logic) is used to represent specifications of system behavior. We use this approach to model the landing gear controller from the complete implementation in Pascal. We also provide temporal algebra interpretations of the specifications made available to us. Finally we perform a number of symbolic analyses on the complete process (controller and plant). This project is a first demonstration of possible uses of these methods and tools and it demonstrates that these methods and tools scale to problems of a non trivial size, i.e. of the size found in complex system designs such as the JAS 39.

1 Introduction

The interest in discrete event systems (DEDS) has increased during the last years, due to the lack of methods and tools that are capable to handle the complexity of problems and tasks present in industry today. To explore the usefulness of symbolic and algebraic methods, we use polynomials over finite fields (see section 2) applied to DEDS with industrial sized complexity: The landing gear controller (LGC) of the Swedish fighter aircraft JAS 39 Gripen.

This paper gives an overview of the project[1] of doing static and dynamic analysis on the behavior of the LGC. (See also [8].) This was made possible by modeling the LGC by a polynomial, i.e. compiling the Pascal implementation of the LGC to a polynomial relation. For a complete description of this project see [3, 6, 7, 4].

2 The Polynomial Framework

Quantities and relations in DEDS are of a finite nature and can therefore be represented by finite relations. These relations are in turn represented mathematically by polynomials over finite fields $\mathbb{F}_q[Z]$, i.e. polynomials of variables in the set Z with coefficients from a finite field $\mathbb{F}_q$. By further restricting the class of polynomials we construct a quotient polynomial ring (see [3]) that gives a one to one correspondence between polynomials and relations as well as a compact representation of the relations.

The computational framework used for manipulating polynomials is based on *binary decision diagrams* (BDD), which give a powerful representation as well as fast computations which allow us to manipulate rather complex systems.

For more information of polynomials over finite fields and its tools see the tutorial paper [5].

3 System Description

The purpose of the LGC is to perform maneuvers of the landing gears and the corresponding doors which enclose the gears in retracted position, see figure 1. The controller is a software process that interacts with 5 binary actuators, 30 binary landing gear sensors, 2 binary pilot signals, and 5 integer mode signals from other subsystems in the aircraft, see figure 2.

The only formal description of the controller available to use was the actual implemented 1200 line Pascal code.

[1]This work was supported by the Swedish Research Council for Engineering Sciences (TFR) and the Swedish National Board for Industrial and Technical Development (NUTEK), which is gratefully acknowledged.

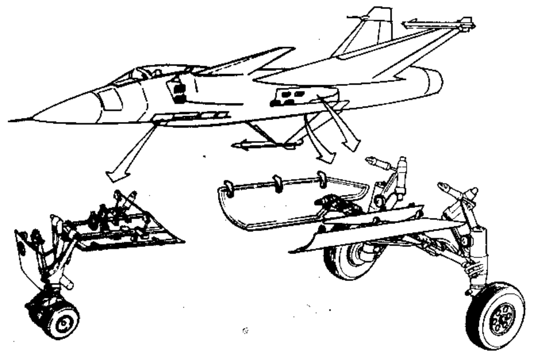


Figure 1: The fighter JAS 39 Gripen.

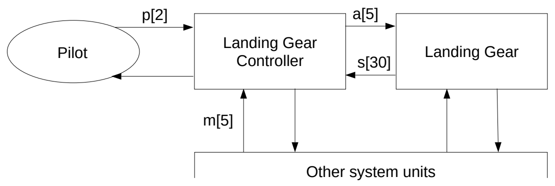


Figure 2: The landing gear system (number of signals in brackets).

4 Modeling

As mentioned in the introduction we build a polynomial model from the implemented Pascal code. To be able to represent the system by polynomials the system must be a DEDS with finite state space. Therefore some restriction on Pascal are needed.

4.1 Restrictions in the Modeling

The allowed data types are integer and boolean. The integer range used is $\{0, \dots, 15\}$, which is enough to represent all enumerable variables in the controller code. The controller code also makes use of linear arrays and abstract data types. It is possible to automatically represent these data types by integers and booleans, but in this case it has been done by hand.

Some of the Pascal primitives have also been excluded. For a list of allowed primitives, see table 1. For code primitives such as `FOR`-loops and the `OTHERWISE` statement in conditionals, a manual translation was made where the `FOR`-loop was rewritten as a sequence of code (loop unrolling, see e.g. [1]) and `OTHERWISE` replaced by explicit arguments.

Timer variables and time conditions in the code have been replaced by binary state variables (flip flops) and corresponding input signals. A time condition becoming true in the original code corresponds to the timer input signal triggering the state variable. Once triggered, the state variable will be true until there is an explicit timer reset.

Comment	Syntax	Domains
Type	`BOOLEAN`	$\mathbb{B}$
	`INTEGER`	$\mathbb{I}$
Arithmetic expr.	`+`	$\mathbb{I}^2 \to \mathbb{I}$
	`-`	$\mathbb{I}^2 \to \mathbb{I}$
Relational expr.	`>`	$\mathbb{I}^2 \to \mathbb{B}$
	`<`	$\mathbb{I}^2 \to \mathbb{B}$
	`=`	$\mathbb{I}^2 \to \mathbb{B}$
	`NOT`	$\mathbb{B} \to \mathbb{B}$
	`AND`	$\mathbb{B}^2 \to \mathbb{B}$
	`OR`	$\mathbb{B}^2 \to \mathbb{B}$
Control	`IF THEN ELSE`	
	`CASE OF`	
	`BEGIN ... END`	
Miscellaneous	`:=`	$\mathbb{I}$ or $\mathbb{B}$
	`VAR`	$\mathbb{I}$ or $\mathbb{B}$
	`PROGRAM`	
	`PROCEDURE`	
	`FUNCTION`	

Table 1: Allowed Pascal primitives. $\mathbb{I}$ and $\mathbb{B}$ stands for integer and Boolean respectively.

From the code module we have excluded the procedures concerning alarm handling and pilot information since they do not affect the other procedures in the controller directly. Since we have not had access to all values of signals from other units in the aircraft we have also defined some new input signals that are aggregations of the unknown signals.

4.2 Input, Output and State Variables

The landing gear controller code is one part of the software loop in the aircraft system. This means that the state of the code is stored until next iteration of the code. If we want to write the system as

$$x^+ = f(x, u), \quad y = g(x, u) \tag{1}$$

we must determine which variables correspond to system state x, next state x^+, input u and output y. The equations in (1) can be represented by a block diagram as in figure 3, where u, y and x are vectors for input, output and state variables respectively. Any part of the code (a single primitive or a complete program) can also be regarded as a function which computes and assigns values to output variables depending on input variables, see figure 4. If we compare figures 3 and 4, we find that the state variables are equal to the variables in the set $Inputs \cap Outputs$. In words we say that if there is an input variable which is reassigned in the code, it should be considered as a state variable.

Temporary variables which are neither input nor

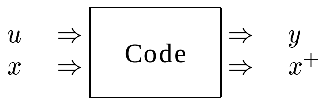


Figure 3: Signal interaction with the code.

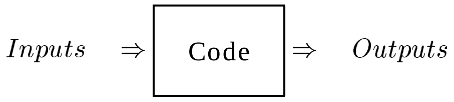


Figure 4: Input-output view of the code.

output variables can be omitted in a model like the one in figure 3, since we are only interested in the output behavior of the system. Still, the compiler must use the temporary variables in the code to compute the relations between input and output variables.

4.3 Translating Pascal to Polynomial Relations

The polynomial model is denoted $M(z, z^+)$, where z and z^+ are the system variables[2] for present and next time instant respectively.

The translation from Pascal to Boolean expressions[3] follows the control flow graph of the program. The value of each program expression is determined by the current values of symbols and the actual program expression, i.e. the compilation function is of the form:

$$\Phi : \mathit{Pascal} \times \mathit{State} \to \mathit{State}$$

We store the current state of the program as a symbol table of the form:

$$\sigma = \{v_1 \mapsto e_1, \ldots, v_n \mapsto e_n\}$$

where each v_i is a variable or symbol and each e_i is a Boolean expression of input variables or the symbol $\perp$ indicating undefined values. The symbol table σ is initiated by variables that acts as place holders for the input, and by $\perp$ for the output variables. The symbol table is then updated by traversing the control flow graph of the Pascal code.

Suppose we have the Pascal expression

$pe =$
```
IF q THEN
  y1 := c
ELSE
  BEGIN
    y1 := d;
    y2 := e
  END;
```

[2]Input, state and output variables.

[3]Boolean expressions are essentially polynomials over the field $\mathbb{F}_2$.

with the initial symbol table

$$\sigma = \{q \mapsto q^*, c \mapsto c^*, d \mapsto d^*, e \mapsto e^*, \\ y1 \mapsto y1^*, y2 \mapsto y2^*\}$$

we will get

$$\sigma^+ = \Phi(pe)\sigma = \{q \mapsto q^*, c \mapsto c^*, d \mapsto d^*, \\ e \mapsto e^*, y1 \mapsto (q^* \wedge c^*) \vee (\neg q^* \wedge d^*), \\ y2 \mapsto (q^* \wedge y2^*) \vee (\neg q^* \wedge e^*)$$

The final Boolean relation is computed from the final symbol table

$$\sigma_{\mathit{final}} = \{x^+ \mapsto f(x, u), y \mapsto g(x, u)\}$$
$$M(z, z^+) = x^+ \leftrightarrow f(x, u) \wedge y \leftrightarrow g(x, u)$$

where $z = [x, y, u]$.

The compilation of the LGC pascal code is performed by a compiler package in *Mathematica* developed by the authors. The resulting relation for the LGC has 26 state variables and the relation $M(z, z^+)$ has 105 variables altogether. The size of the relation is approximately 320 000 nodes as a BDD and takes approximately 35 minutes to compute on a regular workstation.

5 Analysis – Verification

By analysis we mean that, given a polynomial model $M(z, z^+)$, we answer questions about the possible behavior of this model. There are two main types of analysis:

One Step Analysis where we look at a single time step of the system dynamics.

Multiple Step Analysis where we look at arbitrarily many time steps of the system dynamics.

It is important to note that the underlying algebra machinery that supports multiple step analysis has to deal with a far more complex situation than what is required in the one step analysis case.

We can use *temporal algebra* to formulate multiple step analysis questions (and, of course, as a special case of this, one step analysis questions).

5.1 One Step Analysis

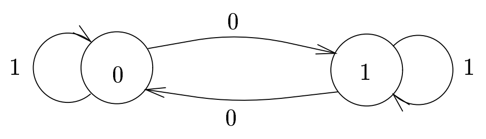
By one step analysis we mean questions that can be resolved as equation systems of the form:

$$M(z, z^+) \wedge Q_1(z) \wedge Q_2(z^+) = 0 \qquad (2)$$

where $M(z, z^+)$ is the process description and $Q_1(z)$ and $Q_2(z^+)$ are restrictions on z and z^+ respectively.

The analysis consists of solving the system of equations or to prove that no such solution exists.

Example 1 Suppose we use the process



and want to know if there is a transition from 0 to 0? If we formulate this in algebraic terms we get

$$M(x,x^+) \wedge Q_1(x) \wedge Q_2(x^+) = (x_1^+ + 1 + x_1 + u_1) \wedge x_1 \wedge x_1^+ \\ = 1 + u_1.$$

The solution to $1 + u_1 = 0$ is $u_1 = 1$, which means that there is a transition from 0 to 0 for the input $u_1 = 1$. Unfortunately, solving Boolean equations is a fairly tough problem in general. In fact it is impossible to have a solution algorithm that behaves reasonably well on all polynomial equations. Partly, this is due to the potential size of the output, i.e. if we have polynomials in n variables over some finite field $\mathbb{F}_p$, there are p^n possible solutions, and in the Boolean case 2^n possible solutions. It is of course impossible to build an algorithm that works faster than it can output its answer. However, even if we formulate the restricted problem of only telling us whether or not there is a solution (without providing us with such a solution), the problem is in all likelihood almost as hard, i.e. if $NP \neq P$.

The results quoted above states that if we are able to solve all possible equations in n variables, then the worst time complexity cannot be reasonable (polynomial in n). In practice one can deal with this problem by using methods that avoid the worst case complexity as often as possible. This is done by using BDDs with a good variabel ordering.

5.2 Multiple Step Analysis

By multiple step analysis we mean analysis questions that take an arbitrary number of time steps of the system dynamics into account. An example is the set of states that are reachable in zero or arbitrarily many steps from some initial state. Given a process model $M(z, z^+)$ we can compute the set of states reachable in k steps or less from some initial set of states, described by $I(z) = 0$, as $R_k(z)$:

$$\begin{aligned} R_0(z) &:= I(z) \\ R_{k+1}(z) &:= R_k(z) \vee (\exists \tilde{z}\ (R_k(\tilde{z}) \wedge M(\tilde{z}, z))) \end{aligned} \quad (3)$$

Since we are dealing with finite state systems this iteration will reach a fixed point, i.e. $R_{d+1}(z) = R_d(z)$ for some finite d. We will give some comments regarding this computation:

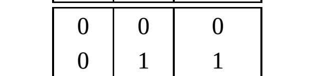
- The d for which we reach the fixed point above is the *depth* of the system. An interpretation of this is that in a maximum of d steps we can reach any reachable state in the system. In general, if we have n binary system variables ($z = [z_1, \ldots, z_n]$) then the maximal possible depth is 2^n. In most engineering applications the depth of a system seems to be far less than the maximal possible depth, but a simple process such as an n bit counter is one that has maximal depth. One can also compare this to an n state linear system, where it is possible to reach any reachable state (from the origin) within n steps.

- In order to compute the set of reachable states (in arbitrarily many steps) we need to solve an equation after each iteration, i.e. when we check whether or not $R_k(z) = R_{k+1}(z)$. As mentioned, this problem is NP complete so we need to solve an NP complete problem in each iteration.

- We need to perform some form of data reduction in each step above, otherwise we will quickly overflow all available memory for all except trivial processes. By using BDDs we always get data reduction in every step.

- In theory we could compute the set of reachable states by using a simulation routine. However in practice this seems quite infeasible as we would have to keep track of all the reached states and then re-initiate the simulation from each of those states until we cannot reach new states any more. Even in moderately complex systems this is hardly feasible, and in the landing gear controller we have in the order of $10\ 000$ reachable states out of 2^{26} potential reachable states. Hence running a few simulation scenarios usually says very little of the system behavior in general.

Example 2 Suppose we have the simple system below:



We can obtain a polynomial model by e.g. introducing two binary state variables x_1 and x_2 and encoding the states as follows

$\mathbf{x_1}$	$\mathbf{x_2}$	**State**
0	0	0
0	1	1
1	0	2

Using this encoding we get the polynomial model

$$M(x,x^+) = ((\neg x_1 \wedge \neg x_2) \wedge (\neg x_1^+ \wedge x_2^+)) \vee \\ ((\neg x_1 \wedge x_2) \wedge (x_1^+ \wedge \neg x_2^+))$$

We can now compute the set of reachable states from state 0 as

$$\begin{aligned} R_0(x) &:= I(x) = \neg x_1 \wedge \neg x_2 \\ R_1(x) &:= ((\neg x_1) \wedge (\neg x_2)) \vee ((\neg x_1) \wedge x_2) \\ R_2(x) &:= ((\neg x_1) \wedge (\neg x_2)) \vee ((\neg x_1) \wedge x_2) \vee (x_1 \wedge (\neg x_2)) \\ R_3(x) &:= ((\neg x_1) \wedge (\neg x_2)) \vee ((\neg x_1) \wedge x_2) \vee (x_1 \wedge (\neg x_2)) \end{aligned}$$

We reach a fixed point for $k = 2$ steps, i.e. in two steps we can reach any reachable state. This can readily be seen from the graph above as well.

In this example we could not have found out that 2 is a reachable state by just one step analysis of $M(x, x^+)$ and the initial state information. In some cases this is important, since some undesirable action might be performed by the controller if it ever reaches state 2. There are a multitude of other types of multiple step analysis that are possible and many of them are related to the idea of reachable states, either backward or forward in time.

5.3 Temporal Algebra and Verification

Since many specifications are written in something close to natural language, we could greatly simplify our analysis task if we could more or less directly translate this to a formal specification. In this application we have used *temporal algebra* [3] (or temporal logic since we use the binary Boolean algebra, see e.g. [2]) to achieve this task. In table 2 some of the most common temporal algebra constructs are given. The expression “$Q(z)$ holds” should be interpreted as $Q(z) = 0$ in terms of polynomials. The *verification* is in general a multiple step analysis. For each temporal algebra expression $S(z)$ and process model $M(z, z^+)$ we compute the set of states from which the temporal algebra statement becomes true.

Example 3 Consider the process from example 2. We wish to verify the specification:

> ”We should always be able to reach the safe state 2 as the next state.”

In terms of temporal algebra this becomes:

$$\text{EX}[x_1 \wedge \neg x_2].$$

The verification is the computation

$$\begin{aligned} &\mathit{Verify}(M(x,x^+), \text{EX}[x_1 \wedge \neg x_2]) = \\ &\quad = \exists x^+ M(x,x^+) \wedge (x_1^+ \wedge \neg x_2^+) \\ &\quad = (\neg x_1) \wedge x_2. \end{aligned}$$

As expected this returns the state 1 in its encoded form, since this is the only state from which we can reach 2 in one step.

Temporal Algebra	**Natural Language**
$Q(z)$	$Q(z)$ holds in the initial state.
EX[$Q(z)$]	$Q(z)$ can hold in the next time step.
EU[$Q_1(z), Q_2(z)$]	$Q_1(z)$ will hold for finitely many steps and then $Q_2(z)$ can hold.
EF[$Q(z)$]	$Q(z)$ can hold at some future time.
EG[$Q(z)$]	$Q(z)$ can hold at all future times, i.e. from this point onwards.
AX[$Q(z)$]	$Q(z)$ must hold in the next time step.
AU[$Q_1(z), Q_2(z)$]	$Q_1(z)$ will hold for finitely many steps and then $Q_2(z)$ must hold.
AF[$Q(z)$]	$Q(z)$ must hold at some future time.
AG[$Q(z)$]	$Q(z)$ must hold at all future times, i.e. from this point onwards.

Table 2: Some of the most common temporal algebra constructs.

Suppose that we have the process and an initial state specified, then the above temporal algebra formula would be verified iff the returned set of states was a superset of the reachable states, i.e. we could reach 2 from every reachable state. For the system above this is clearly not the case if our initial state is 0, since the set of reachable states is $\{0, 1, 2\}$. Generally this extra level of reasoning is built into the verifier.

There are some remarks to be made regarding temporal algebra expressions and the verification of these:

- The constructs $P(z)$, EX[$P(z)$] and AX[$P(z)$] essentially denotes what we have termed one step analysis above.
- The remaining constructs enable multiple step analysis as seen from the user perspective. From the software tools perspective these constructs require the same type of fixed point computations as in the reachability analysis above.
- The temporal algebra statements can be *nested* with themselves as well as ordinary Boolean operations and thus provides great flexibility in expressing specifications.

Example 4 As an example of the flexibility of temporal algebra let us assume that we want to express the following specification:

> “If A can happen then B must never happen.”

Let $Q_A(z)$ be a polynomial describing the condition for the event A, and $Q_B(z)$ a polynomial describing the condition for the event B. We can then express the specification as

$$\text{EF}[Q_A(z)] \to \text{AG}[\neg Q_B(z)]. \tag{4}$$

For more details regarding temporal algebra (or temporal logic), see [2, 3].

5.4 Analysis of the Controller

We use the relation $M(z, z^+)$ to analyze the LGC behavior in a number of ways. First we compute the set of reachable states in the LGC. This set is represented algebraically by a relation $R(x)$. The number of reachable states turns out to be 10 015 which is far below the possible number which is $2^{26} \approx 10^8$. We can restrict the original relation through

$$\hat{M}(z, z^+) = R(x) \wedge M(z, z^+) \wedge R(x^+)$$

which gives a significantly simpler relation.

The one step analysis of $\hat{M}(z, z^+)$ is performed by adding constraints $P(u)$ to the inputs of the LGC, and then analyze what effect this gives to the outputs. The constraints $P(u)$ can e.g. be used to exclude certain unrealistic input combinations. The static analysis performed on the LGC gave corresponding results as obtained by SAAB (developer of JAS39).

Results on dynamic closed loop analysis is not available yet. However we use the same tools as to compute the set of reachable states. The specifications of the behavior are represented by temporal logic expressions, used together with the model to compute e.g. the set of behaviors that might reach a forbidden state in the future.

6 Conclusions

In this case study we have given an example of how one may verify a discrete dynamic control system by building a model of the whole process. In this work we have used the Boolean field to represent the polynomial model $M(z, z^+)$. We can also build a simple model of the function specification $S(z)$ using temporal algebra. Using the closed loop system model $M(z, z^+)$ and the specification $S(z)$ we can then either *verify* or *falsify* the system behavior w.r.t. the specification. In case we falsify the system behavior we can also generate a sequence of inputs that exhibits the failing behavior. This can then be independently verified in a system simulator and the error should be characterized well enough for modification of the controller.

The developed methods and tools allow us to analyze industrial scale discrete systems. In particular this allows us to prove (or disprove) that the system behaves according to its specification. The system should be automatically translated from an internal model description language to the polynomial format, which is suitable for analysis. This procedure eliminates potential discrepancies between documents and the actual system. For dynamic systems, dynamic analysis will ultimately be needed and hence an algebraic computation engine that can handle dynamic analysis is necessary.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244–63, April 1986.

[3] Roger Germundsson. *Symbolic Systems - Theory, Computation and Applications*. PhD thesis, Linköping University, September 1995.

[4] Roger Germundsson, Johan Gunnarsson, and Jonas Plantin. Symbolic algebraic discrete systems - applied to the JAS 39 fighter aircraft. Technical Report LiTH-ISY-R-1718, Department of Electrical Engineering, Linköping University, S-581 83 Linköping, Sweden, December 1994. Available through ftp at ftp://ftp.control.isy.liu.se/pub/Reports/1995/1718.ps.Z.

[5] Johan Gunnarsson. Algebraic methods for discrete event systems - a tutorial. Submitted to WODES96.

[6] Johan Gunnarsson. On modeling of discrete event dynamic systems, using symbolic algebraic methods. Technical Report LiU-TEK-LIC-1995:34, Dept. of Electrical Engineering, Linköping University, S-581 83 Linköping, Sweden, June 1995.

[7] Jonas Plantin. Algebraic methods for verification and control of discrete event dynamic systems. Technical Report LiU-TEK-LIC-1995:33, Dept. of Electrical Engineering, Linköping University, June 1995.

[8] Jonas Plantin, Johan Gunnarsson, and Roger Germundsson. Symbolic algebraic discrete systems theory - applied to a fighter aircraft. In *34th IEEE Conference on Decision and Control*, pages 1863–1864, 1995.