Symbolic Algebraic Discrete Systems – Applied to the JAS 39 Fighter Aircraft*

Roger Germundsson Johan Gunnarsson

Jonas Plantin

<roger,johan,plantin@isy.liu.se>†

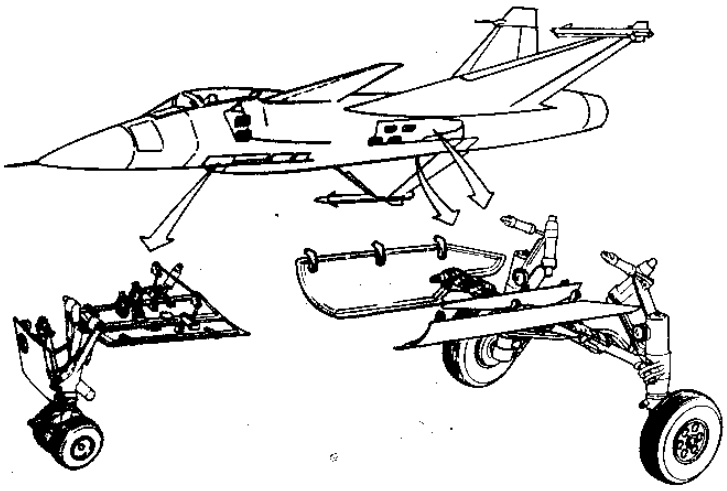
Division of Automatic Control

Department of Electrical Engineering

Linköping University

S-581 83 Linköping, Sweden

31 December 1994



*This report is available through anonymous ftp <joakim.isy.liu.se:/pub/Reports/1994/1718.ps.Z> or by WWW <http://joakim.isy.liu.se/AutomaticControl/Reports/index.html>

†You may also contact all authors through <sads@joakim.isy.liu.se>, where SADS is the acronym for Symbolic Algebraic Discrete Systems.

Abstract

In this document we present symbolic algebraic modeling and analysis techniques applied to the landing gear subsystem in the new Swedish fighter aircraft, JAS 39 Gripen. This is a work in progress report within the larger project COmplex Hybrid SYstems (COHSY) which is a joint project between several academic (Mechanical Engineering, Computer Science and Electrical Engineering) and industrial (Saab Aircraft, Volvo Aero and VOAC) partners. It is also part of a long term effort within the Automatic Control group in Linköping to deal with discrete dynamic systems, mainly for control applications.

Our methods are based on polynomials over finite fields (with Boolean algebra and propositional logic as special cases). 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 a partial documentation as well as 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 (or verifications) on the complete process (controller and plant).

The main results are:

- 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.
- A first demonstration of possible uses of these methods and tools.
- Several interesting avenues of continued research motivated by problems found during this application.

Key Words: Discrete Dynamic Systems, Control, Finite Field Polynomial, Boolean Algebra, Propositional Logic, Binary Decision Diagrams, Temporal Logic, Modeling, Model Compilation, Analysis, Verification, Fighter Aircraft, Landing Gear

Contents

1 Introduction

1.1 Executive Summary

We have modeled and analyzed an existing discrete subsystem of a modern fighter aircraft, the landing gear system on the JAS 39 Gripen (cover illustration). This system was designed and implemented without any formal methods or tools as is usually the current practice for discrete dynamic systems in industry today. We have built a mathematical model of this system and its (natural language) specification and analyzed its behavior w.r.t. to this specification. The main focus has not been on the specific system, but rather on the general methods that can be applied to discrete dynamic systems of industrial size. Some of the tentative conclusions so far are[1]:

- The model representation we use does allow us to analyze processes of a complexity seen in industry.
- Symbolic methods allow the system designer to verify and automate many stages in the design process. This should lead to a faster design cycle as well as more reliable designs.
- Having a system description given as an implementation is far from an ideal situation. In order to get the full benefit of symbolic methods one has to integrate methods and tools tightly into the design process.

1.2 Outline

Section 1: The remainder of this introduction provides background material of the COHSY project as well as goals and status on symbolic algebraic discrete systems theory as found in the Automatic Control group.

Section 2: Provides a brief technical introduction to the methods and tools employed in this project.

Section 3: A description of the studied system, i.e. landing gears and the controller.

Section 4: Model and analysis based on a partial documentation that was made available at an early stage.

Section 5: Model and analysis based on the full Pascal implementation of the controller.

Section 6: Some technical conclusions from this project.

Section 7: The people and organizations that made this work possible.

[1] The work is still in progress

1.3 Background

This is a work in progress report within the larger project COmplex Hybrid SYstems (COHSY) which is a joint project between several academic (Mechanical Engineering, Computer Science and Electrical Engineering) and industrial (Saab Aircraft, Volvo Aero and VOAC) partners. It is also part of a long term effort within the Automatic Control group in Linköping to deal with discrete dynamic systems, mainly for control applications.

We wanted to provide an acid test to the methods and tools, for discrete dynamic systems, developed over a longer time period by the first author. Our industrial partner (Saab Aircraft) wanted more predictable and reliable development methods for their discrete subsystems. The landing gear was identified as one such subsystem where substantial development effort had been spent and only limited formal verification had been attempted. The mutual benefits from the project makes it a good example of interaction between industrial applications and academic research.

1.4 Goals

The goal of this subproject is to investigate the use of exact formal methods in the (industrial) design process of mainly discrete systems. The design process is roughly grouped in the following categories:

Modeling: How to obtain a mathematical model of the process that has the same behavior as the real process.

Analysis: Analyze the behavior of the model.

Design: Modify the behavior of the process, through the use of a controller or supervisor.

Implementation: Generate an implementation that has the same behavior as the model of the controller.

Our goal is to capture all of these categories in such a way that it is relevant in an industrial setting. In this report we see examples from the first two categories.

1.5 Completed Work

1.5.1 Current Tools and Methods in Industry

During 1994 we have investigated the tools and procedures used by our industrial partners, see [5]. From this study we have concluded that (in connection with discrete systems):

- The available tools offer only limited capabilities, i.e. usually only simulation or only various types of static analysis of dynamic systems.

- The use of formal or symbolic methods is very limited, i.e. has not been integrated into the development process and the methods employed have very limited scope. This is of course strongly connected with the limited capabilities of the available tools.

In particular this means that there is a great potential for speeding up the development process as well as improving the quality of the end result through the use of symbolic methods.

1.5.2 Current Tools and Methods in Research

We have over a number of years developed the mathematics and algorithms to deal with discrete dynamic systems. These generally fall into the same categories as we identified above (modeling, analysis, design and implementation) and the symbolic representation and manipulation is based on polynomials over finite fields. Some of the crucial issues that have been addressed are:

- To have a framework that allows us to rigorously formulate many analysis and design problems both for static and dynamic discrete systems.
- To have a framework that allows us to do computational formulations of analysis and design problems that does scale to industrial size systems. These computations are symbolic (in most cases algebraic) as opposed to numeric and are generally *complete* in some sense.

The first item means that there is a solid foundation on which methods and algorithms are built. We essentially cover all finite state processes. The second item means that we can usually perform analysis and designs for rather complex processes, i.e. processes having many system variables[2]. The second item also indicate that the computations usually also work as a proof, i.e. are complete. Some examples:

- If we have compiled a model from some description language (e.g. Pascal in this project) into our polynomial framework, then the polynomial model really has exactly the same behavior as the original description. Contrast this with a manual or semi-manual approach where we could only say that the *model* has a certain behavior since the manual procedure could have introduced a number of discrepancies.
- If we have verified that a behavior is impossible in a discrete system then it cannot occur. Contrast this with the result gained from extensive testing or simulation where we could only say that the undesired behavior is unlikely to occur, depending on the amount of testing and simulation.
- If we have computed that there is no controller that satisfies a given set of constraints, then there really exists none. Contrast this with a manual or heuristic approach where you could only say that the efforts so far has not produced a controller.

[2] Equivalently: Processes having many possible inputs, states and outputs.

For the actual methods and framework employed in this case study, see section 2.

1.5.3 Industrial Case Studies

As part of this project we wanted to test the feasibility of using symbolic methods for a process of industrial size. During 1994 we have studied a discrete industrial process: the landing gear controller on the JAS 39 Gripen fighter aircraft.

Judging by the variable declarations in the landing gear module of the implemented code, we would have to handle some 30 variables, half of which are Boolean. However, since several variables had been declared outside the module it turned out that the process is fairly complex, with some 80 variables, of which 66 are Boolean. In addition, the fact that the code module did not contain all information needed, made the modeling part of the project somewhat more complicated. The landing gear process is described in greater detail in section 3.

We have built two models based on either a partial documentation[3], see section 4, or based on the actual implementation of the controller in Pascal, see section 5. We have also built formal models of the system specification. The dynamic models are internally represented as polynomial difference equations whereas the specification is in terms of temporal algebra. See section 2 for more details on the methods employed. See sections 4 and 5 for these methods applied to the landing gear process.

1.6 Future Work

Our first goal is of course to complete the current case studies, but there are several issues related to this that could be improved. All of the issues below relate to industrial scale problems. The issues are presented without any specific ranking in order to solicit comments from our industrial partners.

1.6.1 Modeling

We need more research on model description languages aimed at symbolic analysis (e.g. verification) rather than simulation or code generation. There are several candidate languages to examine, some are international standards and some are industry specific model description languages. Furthermore we need to better understand the relation between model complexity, as seen in some model description language, and model complexity in the polynomial representation. This is of crucial importance when doing symbolic analysis and design. In doing this industrial example we have uncovered quite a number of such principles, but the whole problem would greatly benefit from a more focused study.

[3] We had some problems obtaining the full system description initially. We also needed a smaller problem to develop and evaluate our tools.

1.6.2 Analysis

In the analysis part there are a number of important issues concerning complexity. In particular in performing the dynamic analyses (e.g. dynamic verification) there are several equivalent computational formulations that have wildly different space and time complexities.

1.6.3 Design

We should also start looking for potential applications regarding *design* and *implementation* categories. By automatic design we mean that, using some more abstract version of our specification (perhaps as a temporal algebra expression) and a plant model, we wish to automatically compute a controller which combined with the plant achieves the specification. Various restrictive versions of this problem have been solved in the control community and by the authors of this report, but we have not attempted this on a large industrial problem.

1.6.4 Implementation

Given that we have a model described as a polynomial difference equation (perhaps a controller computed through the design procedure above) we would like to generate alternative representations of this. The alternative representation could be another model description language that is used by other design teams or it could be an executable implementation meant to run on some microprocessor. Finally it could also be a document. All of these examples are in effect the reverse of the modeling problem and of definite industrial relevance since they would allow these tools to interact with other tools, generating implementations and documentations that are guaranteed to correspond to the design.

2 Methods and Tools

In this section we briefly describe the methods and software tools used in the industrial case studies. In the case studies we focus on modeling and analysis and hence these are the aspects covered below.

2.1 Modeling

By modeling we essentially mean building a mathematical model. The preferred mathematical model type for our purposes is a polynomial model. In practical engineering work however, several other model descriptions are used and we thus have to translate (or compile) from these descriptions to the desired form needed in order to do analysis.

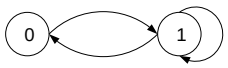
2.1.1 Polynomial Models

Polynomial models are used for computational purposes in subsection 2.2 below and are of the form:

$$M(z, z^+) \tag{1}$$

where $z = [z_1, \ldots, z_l]$ are all the the *system variables* typically classified as *input*, *output* and *state*, but other groupings could be possible. Furthermore z^+ denotes the value of z one time instant into the future, i.e. if we use some index to indicate time we get $z(k)^+ = z(k+1)$. Finally M is a polynomial in z and z^+. In particular we will use the finite field $\mathbb{F}_2 = \{0, 1\}$ in this application. This means that our polynomials are essentially Boolean polynomials. The main reason is the application itself and the (large scale) tools (see subsection 2.3) we have developed so far.

Example 2.1 *Consider the simple dynamic system given by the graph:*



We can interpret this as a system that can be in either of two states: 0 *and* 1*. Furthermore in each (discrete) time instant the system will transition along one of the paths extending from the current state. We have not specified how the system chooses path for a transition, but following standard control practice we could add an external input* v *(and probably call it* noise*) that does the choice for us. This is however irrelevant for the example.*

By encoding the states in the graph above in a Boolean vector and then into Boolean expressions we get: (the first encoding is trivial in this case)

$$0 \mapsto 0 \mapsto \neg x, \quad 1 \mapsto 1 \mapsto x$$

Using this encoding we can describe the process by the relation:

$$M(x, x^+) := x \vee x^+$$

where $\vee$ *denotes* or*.*

When we have a simulation model or an actual implementation the polynomial model typically is of the form:

$$x^+ = f(x, u), \quad y = g(x, u) \tag{2}$$

where $x = [x_1 \ldots, x_n]$ are the state variables, x^+ the next state variables (at the next time instant), $u = [u_1, \ldots, u_m]$ are the input signals and $y = [y_1, \ldots, y_p]$ are the output signals. The function $f(x, u) = [f_1(x, u), \ldots, f_n(x, u)]$ is the state transition function and $g(x, u) = [g_1(x, u), \ldots, g_p(x, u)]$ is the output function. This is just a special case of the model in equation (1) since if we let $z = [x; u; y]$ we get:

$$M(z, z^+) := (x^+ = f(x, u)) \wedge (y = g(x, u)) \wedge (y^+ = g(x^+, u^+)) \tag{3}$$

where $\wedge$ denotes *and*.

It is possible to prove that we can represent *any* finite state process with a polynomial model.

2.1.2 Model Description Languages

For most practical purposes, such as model input, model editing and model documentation, the representation in equation (1) is impractical. Rather some form of model description language should be used, possibly with accompanying graphical representation and graphical editor. There are a myriad of model description languages available and most tool vendors seem to have several proprietary versions just for good measure. There are however a couple that have been accepted as international standards: VHDL [7] for integrated circuits, SDL [2] for telecom applications and Grafcet [3] for sequential control. Even if these standards have been developed within standardization bodies for specific disciplines they are useful outside these disciplines.

If our process is described in terms of a model description language there should be a well defined behavior associated with that process, i.e. given input and initializations it should be possible to generate unique outputs.[4] If the model is a finite state model, we should be able to translate this description to an equivalent polynomial model, in the sense that they have the same behavior. In practice it is usually both simpler and safer to write a general translation routine for the whole model description language or a subset thereof, i.e. a *compiler*[5]. The model compiler can then be seen as a function:

$$\phi : \text{Model Description Language} \rightarrow \text{Polynomial Model} \tag{4}$$

where the behavior is preserved. This is in effect what we have done in section 5, where the model description language is Pascal and the polynomial model is a Boolean model. However since a Pascal program can potentially have infinite state we have to restrict this compilation to a subset of Pascal. See section 5 for more details.

2.2 Analysis – Verification

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

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

Dynamic Analysis where we look at arbitrarily many time steps in the system dynamics. This allows us to verify properties that depend on the dynamic behavior of the process.

[4] This of course assumes explicit models, but all the mentioned cases conform to that.

[5] We will use compiler and translator interchangeably in the remainder of this document.

It is important to note that the underlying algebra machinery that supports dynamic analysis have to deal with a far more complex situation than what is required in the static analysis case.

We can use *temporal algebra* to formulate mixtures of static and dynamic analysis questions.

2.2.1 Static Analysis

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

$$M(z, z^+) \wedge R_1(z) \wedge R_2(z^+)$$

where $M(z, z^+)$ is the process description and $R_1(z)$ and $R_2(z^+)$ are restrictions on z and z^+ respectively. The actual analysis job is then to solve the system of equations or to prove that no such solution exists.

Example 2.2 *Suppose we use the process model from example 2.1 and pose the question:*

Is there a transition from 0 *to* 0*?*

If we reformulate this in algebraic terms we get:

$$M(x, x^+) \wedge R_1(x) \wedge R_2(x^+) := (x \vee x^+) \wedge (\neg x) \wedge (\neg x^+)$$

In this case we can easily see that there is no solution to the equation above[6]*.*

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 $\mathbb{F}_p$ then 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 a such a solution), the problem is in all likelihood almost as hard. If this was not the case then a whole set of other very hard problems would turn out to be simple as well, since they have been translated into this solvability question. Technically the solvability problem is known to be *NP complete*, see Hopcroft et.al [6][7] for details.

The results quoted above merely states that if you are able to solve all possible equations in n variables, then the worst time complexity cannot be reasonable, i.e. polynomial. In practice one can deal with this problem by using methods that avoids the worst case complexity as often as possible. See subsection 2.3 below for details on how this is done.

[6] Since these are Boolean expressions, a solution is an assignment of values to x and x^+ such that the whole expression is 1 (or true).

[7] Look for *satisfiability*.

2.2.2 Dynamic Analysis

By dynamic analysis we mean analysis questions that take the dynamics into account. Some examples include 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 $I(z)$ as $R_k(z)$:

$$R_0(z) := I(z) \tag{5}$$

$$R_{k+1}(z) := R_k(z) \vee (\exists \tilde{z}\ (R_k(\tilde{z}) \wedge M(\tilde{z}, z))) \tag{6}$$

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

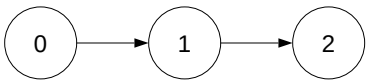
- The least k for which we reach the fixed point above is the *depth* of the system. An interpretation of this is that in a maximum of k steps we can reach any reachable state in the system. In general if we have n binary system variables ($z = [z_1, \ldots, z_n]$) then the maximal possible depth is 2^n. In most engineering applications the depth of a system seems to be far below its 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 $R_k(z) = R_{k+1}(z)$. This means that 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. This means that so called *proof theory* based methods as exhibited in NP Circuit [8] cannot be used in a dynamic setting. See subsection 2.3 for how we have dealt with this problem.

- In theory we could compute the set of reachable states by having 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, e.g. 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 2.3 *Suppose we have the simple system below:*



which could be described by the relation $R = \{(0,1),(1,2)\}$*. We can obtain a polynomial model by first encoding the states* $0,1,2$ *as binary vectors and then to Boolean expressions:*

$$0 \mapsto [0,0] \mapsto \neg x_1 \wedge \neg x_2,\ 1 \mapsto [0,1] \mapsto \neg x_1 \wedge x_2,\ 2 \mapsto [1,0] \mapsto x_1 \wedge \neg x_2$$

Hence 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*:*

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

Hence 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 chose to reduce the size of the intermediary formulas. Suppose we do not do any formula reductions then already* R_1 *(computed according to equation (6)) would be quite large and intangible:*

$$\begin{aligned}
&((\neg x_1) \wedge (\neg x_2)) \vee (((((\neg 0) \wedge (\neg 0)) \wedge ((((\neg 0) \wedge (\neg 0)) \wedge \\
&((\neg x_1) \wedge x_2)) \vee (((\neg 0) \wedge 0) \wedge (x_1 \wedge (\neg x_2))))) \vee (((\neg 1) \wedge \\
&(\neg 0)) \wedge ((((\neg 1) \wedge (\neg 0)) \wedge ((\neg x_1) \wedge x_2)) \vee (((\neg 1) \wedge 0) \wedge (x_1 \wedge \\
&(\neg x_2)))))) \vee ((((\neg 0) \wedge (\neg 1)) \wedge ((((\neg 0) \wedge (\neg 1)) \wedge ((\neg x_1) \wedge \\
&x_2)) \vee (((\neg 0) \wedge 1) \wedge (x_1 \wedge (\neg x_2))))) \vee (((\neg 1) \wedge (\neg 1)) \wedge ((((\neg 1) \wedge \\
&(\neg 1)) \wedge ((\neg x_1) \wedge x_2)) \vee (((\neg 1) \wedge 1) \wedge (x_1 \wedge (\neg x_2)))))))
\end{aligned}$$

Note however, that there are several levels of reduction that could be performed, starting at eliminating all constants up to computing a canonical form.

In this example we could not have found out that 2 *is a reachable state by just static 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 dynamic analysis that are possible and many of them are related to the idea of reachable states either backward or forward in time.

Temporal Algebra	Natural Language
$P(z)$	$P(z)$ holds in the initial state.
EX$[P(z)]$	$P(z)$ can hold in the next time step.
EU$[P_1(z), P_2(z)]$	$P_1(z)$ will hold for finitely many steps and then $P_2(z)$ can hold.
EF$[P(z)]$	$P(z)$ can hold at some future time.
EG$[P(z)]$	$P(z)$ can hold at all future times, i.e. from this point onwards.

Table 1: Some of the most common temporal algebra constructs.

2.2.3 Temporal Algebra and Verification

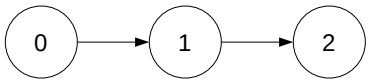
Since many specifications are written in something close to natural language, we could greatly simplify our analysis task if we could more or less directly translate this to a formal specification. In this application we have used *temporal algebra* (or temporal logic since we use the binary Boolean algebra) to achieve this task. In table 1 some of the most common temporal algebra constructs are given. Furthermore there is a set of dual constructs to the ones in table 1 where E is replaced by A having the meaning that we exchange the words *can hold* with *must hold*. A simple instance of this is:

$$\text{AX}[P(z)] \sim P(z) \text{ must hold for all possible next time instants}$$

In this manner we have attempted to interpret parts of the JAS specifications into temporal algebra expressions.

The analysis part, or *verification*, is a mixture of static and dynamic 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 2.4 *Consider the process from example 2.3:*



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:

$$EX[2] = EX[x_1 \wedge \neg x_2]$$

where we have used the algebraic encoding to the right. The actual verification then computes:

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

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

Suppose we now 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. In the case above this is clearly not the case if our initial state is* 0, *since the set of reachable states is* $\{0, 1, 2\}$ *in that case. Generally this extra level of reasoning is of course built into our verifier.*

There are some remarks 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 static analysis above.
- The remaining constructs enables dynamic analysis as seen from the user perspective. From the software tools perspective these constructs require that the underlying algebra machinery supports dynamic analysis since verification of these require the same type of fixed point computations as was seen for the reachability analysis above. One consequence of this is that so called proof theory methods cannot be used to verify these constructs.
- The temporal algebra statements can be *nested* with themselves as well as ordinary Boolean operations and thus provides great flexibility in expressing specifications.

For more details regarding temporal algebra (or temporal logic), see [4].

2.3 Software Tools

This subsection will briefly describe the software tools used in the case studies. Most of this software was developed over a period of several years by the first author. Furthermore the parts described in this document only constitute a small part of the whole software system. The full system will be thoroughly described elsewhere.

The experimental system consists of *Mathematica* [10] code together with externally linked C code for critical operations through the MathLink structured communication protocol. *Mathematica* was chosen for its excellent programming model and plenty of numeric, symbolic and graphical algorithms well integrated into a consistent system.

2.3.1 Modeling

The modeling part follows the general outline provided in subsection 2.1 above. In this case a new model compiler was written (by the latter two authors) for the Pascal subset used in the controller implementation. This is described in section 5 below.

2.3.2 Analysis

The analysis software was written by the first author and makes use of an efficient implementation of Boolean algebra, known as *binary decision diagrams* (BDD), see [1] for details.[8] Using an efficient symbolic algebraic computation engine is crucial if we are to be able to analyze realistically sized examples. As pointed out earlier we cannot utilize a proof theory based computation engine if we want to perform dynamic analysis.

The basic idea used in binary decision diagrams is to rewrite Boolean expressions in a recursive form and reuse common subexpressions, a technique that has been used in compiler optimization for several decades. In the case of Boolean expressions this leads to highly efficient computations in most cases.

Suppose we have a Boolean expression $f(x_1, \ldots, x_n)$, we can then rewrite it using Shannon's expansion formula:

$$f(x_1, \ldots, x_n) = ((\neg x_1) \wedge f(0, x_2, \ldots, x_n)) \vee (x_1 \wedge f(1, x_2, \ldots, x_n))$$

If we continue with this recursively for each of the new functions $f(0, x_2, \ldots, x_n)$ and $f(1, x_2, \ldots, x_n)$ w.r.t. x_2 and then x_3 etc, we obtain:

$$f(x_1, \ldots, x_n) = \neg x_1 \wedge (\cdots \underbrace{(\underbrace{(\neg x_n \wedge \alpha)}_{g^{n-1}_{0,\ldots,0}(x_n)} \vee \underbrace{(x_n \wedge \beta)}_{g^{n-1}_{0,\ldots,1}(x_n)})}_{g^1_0(x_2,\ldots,x_n)})$$
$$\vee$$
$$x_1 \wedge (\cdots \underbrace{(\underbrace{(\neg x_n \wedge \gamma)}_{g^{n-1}_{1,\ldots,0}(x_n)} \vee \underbrace{(x_n \wedge \delta)}_{g^{n-1}_{1,\ldots,1}(x_n)})}_{g^1_1(x_2,\ldots,x_n)})$$

where $\alpha, \beta, \gamma, \delta \in \{0, 1\}$. Furthermore we see that we obtain several subexpressions with progressively fewer variables. In fact all expressions g^i above are Boolean expressions in the variables $\{x_{i+1}, \ldots, x_n\}$. In the case that some of these expressions are equal we should not have to repeat this part more than once and then substitute a reference to this common subexpression. The recursive Boolean expression form above can be visualized as a binary tree, where each node corresponds to the $\vee$ operator and essentially the g^i expressions as subtrees. This is the basis of the name BDD. The number of remaining nodes (or equivalently the number of *different* subexpressions) is a measure of the complexity of the given expression. This is referred to as the number of nodes in section 5. By changing the order in which we expand w.r.t. the various variables we usually get wildly different node counts. The ordering is termed *variable ordering* and plays a significant role in lowering the representational complexity of the landing gear system in section 5.

[8] The first author has designed an implementation for general finite field polynomials, but the actual implementation was not available for this case study.

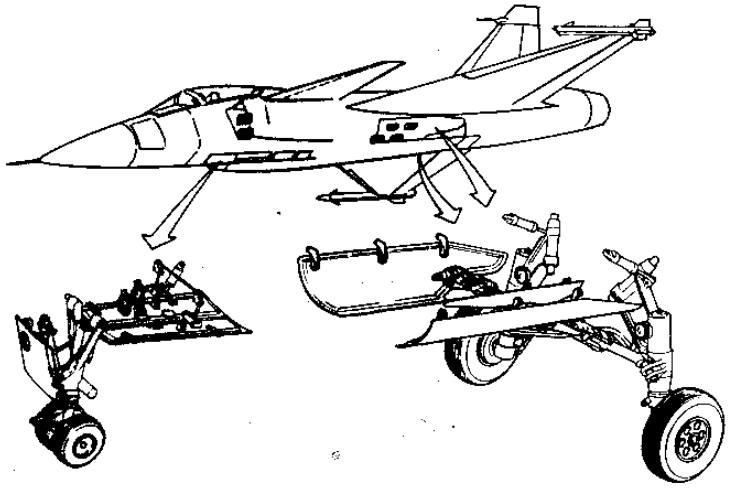


Figure 1: The fighter JAS 39 Gripen.

3 The Landing Gear Process

3.1 Overview

The case studies in sections 4 and 5 concerns the landing gear system on the Swedish fighter JAS 39 Gripen, depicted in figure 1.

The landing gear system consists of a landing gear controller and three landing gears with corresponding doors. A simplified block description of the complete system is shown in figure 2, where the arrows should be interpreted as signal vectors.

The objective of the studies is to apply methods for formal verification on the landing gear controller.

3.2 The Physical System

Besides the fact that the controller itself is discrete, the domains of all actuator and measurement signals are discrete. However the underlying system is continuous and to fully understand the expected behavior of the controller, we need to know how the gears work.

Maneuvering of gears and doors are commanded electrically but actuated hydraulically. There is one hydraulic actuator for each gear as well as one actuator for each door. However, all gears are operated in parallel, as are the doors, i.e. it is not possible to close one door while keeping the other two open. In addition to the valves controlling gears and doors there is a valve that shuts off hydraulic pressure in the system during flight with retracted gears.

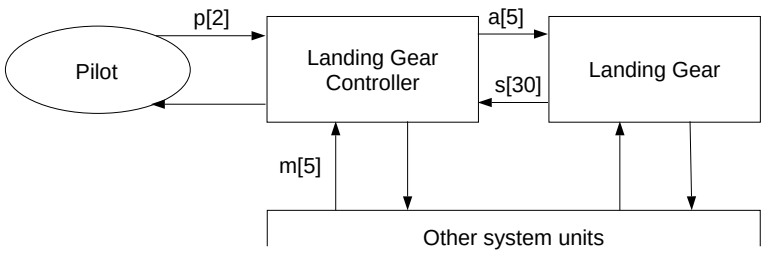


Figure 2: The landing gear system (number of signals in brackets).

There are basically three maneuver types, retraction, extension and emergency extension. Ordinary extension or retraction is commanded by a lever in the cockpit. During emergency extension the control signals are generated by hardware logic and not by the landing gear controller. However when emergency extension is initiated a signal is transmitted to the landing gear controller via other units in the aircraft. In emergency extension mode hydraulic power is not used to lower the rear gears, they are opened by the air drag.

The feedback from the gears to the controller is generated by microswitches positioned on gears and doors. By these switches it is possible to detect certain discrete positions of the doors and gears, e.g. doors open, doors closed, gears retracted and gear extended. In fact, most of the interface between the gears and the controller is discrete. This means that the models of the gears, which we need for dynamic analysis, can be made discrete. The degree of refinement in the gear models will reflect the desired fidelity.

3.3 The Signal Interface

As shown in figure 2 there are basically three kinds of input to the landing gear controller: pilot commands, information from other units in the system and feedback from gears and doors.

The pilot command is detected by a double microswitch positioned on the extension lever. The system information consists of input from various other parts of the aircraft, e.g. power supply, hydraulic supply and motor. As mentioned, the gear feedback comes from microswitches on gears and doors. There are three microswitches on each gear and two on each door and each of the microswitches has two contacts. This makes a total of 30 binary signals from the gears and doors.

A detailed description of all input signals can be found in tables 2, 4 and 5. All signals except m_3 and m_4 are binary. Note that only the signals that will be included in our models of the landing gear controller have been given a short name.

The output from the controller consists of actuator signals, information signals to other aircraft units and signals used for status presentation in the cock-

Signal Name	Short Name	Description
start_landn_panel.ut1	p_1	Extend gears (switch 1)
start_landn_panel.ut2	p_2	Extend gears (switch 2)

Table 2: Pilot input: Commands given by the pilot to control the landing gear.

Signal Name	Short Name	Description
utport_dd00[o_avst_vnt]	a_1	Hydraulic pressure on
utport_dd00[fall_in]	a_2	Retract gears
utport_dd00[fll_ut_st]	a_3	Extend gears
utport_dd00[s_lndst_lck]	a_4	Close doors
utport_dd00[o_lndst_lck]	a_5	Open doors

Table 3: Actuator signals: Commands to the hydraulic system.

Signal Name	Short Name	Description
nodutf_stall	m_1	Emergency extension commanded
motorn_gaur	m_2	Motor is running
fpl_tillst	m_3	Aircraft status
mark_tillst	m_4	Aircraft status
plus28v_kritblk	m_5	An element in the variable so_info.fo.power that signals power down in AFPL
adc_t_afpl8	-	
ess_t_afpl30.fsw.mode	-	
hyd_larm_f	-	
paudrag	-	
subfas_inom_1hz	-	
update_cnt_ok	-	

Table 4: System state: Signals from other units in the aircraft.

Signal Name	Short Name	Description
nosstall_inne	s_1^1, s_1^2	Nose gear retracted
nosstall_ute	s_2^1, s_2^2	Nose gear extracted
nosstall_infj	s_3^1, s_3^2	Weight on nose gear
nosstall_lucka_stangd	s_4^1, s_4^2	Nose door closed
nosstall_lucka_oppen	s_5^1, s_5^2	Nose door open
hstall_h_inne	s_6^1, s_6^2	Right gear retracted
hstall_h_ute	s_7^1, s_7^2	Right gear extracted
hstall_h_infj	s_8^1, s_8^2	Weight on right gear
hstall_lucka_h_stangd	s_9^1, s_9^2	Right door closed
hstall_lucka_h_oppen	s_{10}^1, s_{10}^2	Right door open
hstall_v_inne	s_{11}^1, s_{11}^2	Left gear retracted
hstall_v_ute	s_{12}^1, s_{12}^2	Left gear extracted
hstall_v_infj	s_{13}^1, s_{13}^2	Weight on left gear
hstall_lucka_v_stangd	s_{14}^1, s_{14}^2	Left door closed
hstall_lucka_v_oppen	s_{15}^1, s_{15}^2	Left door open

Table 5: Landing gear feedback: Output from microswitches on gears and doors.

pit. There are five (binary) signals to the actuators which are presented in table 3. The other output signals will not be included in our models and are therefore not described in detail.

4 Modeling and Analysis Based on Documents

This represents a first effort to verify the landing gear controller w.r.t. to its specification. This effort is based on partial documentation of the landing gear controller that was made available at an early stage [9]. The full documentation of the landing gear control software including code was made available at a later stage. The partial documentation represented a good opportunity to fine tune the actual software used for the verification process. In fact many more packages were written and given their first real life test in this initial phase. This effort also represented a first test case for the theories underlying the full version in section 5.

The partial documentation does give a fairly complete picture of the controller for the extension maneuver, whereas the retraction and emergency extension maneuvers are not covered. Since specification available in the partial documentation mainly covers the correct way of changing between the various maneuver types we have used some simple test specifications that allows us to check whether or not we have a reasonable model.

Caveats: There are some major drawbacks with working from a (paper) documentation such as:

- Since these paper documents was not automatically generated we do not know for sure that they correspond to the *true* controller. Hence any bugs found need to be independently verified in the real controller.
- Since these are paper documents we have to manually input their content when building the model, thus potentially introducing some errors.

These drawbacks disappear if we work from an electronic document which is known to correspond to the actual implemented system, perhaps through automatic translation to an implementation or in our case the actual implementation.

4.1 Controller Model

We build a model for the controller in the case of an extension maneuver. This part of the controller only uses a subset of the input and output signals of the complete landing gear controller since many of the issues related to the change of maneuver are dealt with elsewhere. All of the signals in this controller are binary.

Input: The microswitch information from the gears and doors, i.e. $s_1, \ldots, s_{15}$ (where $s_i = s_i^1 \wedge s_i^2$). System state information, i.e. $m_1, \ldots, m_5$. Certain time outs, i.e. the controller makes use of timers in order to satisfy certain time based constraints in the specification.

Output: Actuator control signals, i.e. $a_1, \ldots, a_5$.

State: The internal controller state is based on 7 binary variables $x_0, \ldots, x_6$.

The general form of the resulting controller is:

$$x^+ = f(x,u), \quad y = g(x), \quad x(0) = [1, 0, \ldots, 0]$$

where $u = [s_1, \ldots, s_{15}; m_1, \ldots, m_6, to_1]$ is the input and $y = [a_1, \ldots, a_5]$ is the output and $x = [x_0, \ldots, x_6]$ is the internal state. We can generate a relational form of this controller through:

$$C(z, z^+) := (x^+ = f(x,u)) \wedge (y = g(x)) \wedge (y^+ = g(x^+)) \tag{7}$$

where $z = [x; u; y]$ are all the *system variables*.

4.2 Plant Model

In order to get a closed loop system, we also need to build a model of the system with which the controller interacts. At some level of abstraction the landing gears could be modeled as a continuous process by using, e.g. Newton's laws

etc. However, we interact with this process through a discrete interface in the sense that the actuator signals are either on or off and the output signals (mainly from microswitches) are also either on or off. We can then build a purely discrete version of the plant. Furthermore we can adjust the level of detail in the plant model to reflect aspects of the specification or the real plant.

In this initial phase we use simple static models, see subsection 5.2 for a simple dynamic plant model. This allows us to examine how sensitive the landing gear control system is w.r.t. failures in the landing gear plant (or some of its sensors).

- The trivial plant. In this model any sequence of outputs can be generated.

$$P_1(z, z^+) := 1 \tag{8}$$

- Noting that *if* the plant operates correctly (without sensor failure) certain combinations of microswitches cannot give an output simultaneously. We then obtain static constraints on $s_1, \ldots, s_{15}$, i.e. $\pi_1(s)$ for some π,

$$P_2(z, z^+) := P_1(z, z^+) \wedge \pi_1(s) \wedge \pi_1(s^+) \tag{9}$$

- Similarly *if* the remainder of the aircraft operates correctly then we could only be in one of the aircraft modes (described in the partial documentation) simultaneously, i.e. if we denote these modes by ν_i, only one of ν_1, ν_2, ν_3 could be true simultaneously or $\pi_2(\nu) = EQ_1(\nu_1, \nu_2, \nu_3)$ where EQ_1 denotes a predicate that is true iff exactly one of its arguments is true[9]. If we add these constraints to plant P_2 we get a strictly less expressive model in the sense that there are fewer input output sequences possible in P_3 (below) than in P_2

$$P_3(z, z^+) := P_2(z, z^+) \wedge \pi_2(\nu) \wedge \pi_2(\nu^+) \tag{10}$$

- By analyzing the possible behaviors in plant P_3 we see that there does seem to be a time out (to_1) responsible for some the failing behaviors. Hence we add a no time out model to check if the system still can fail when there is *no* time out.

$$P_4(z, z^+) := P_3(z, z^+) \wedge (\neg to_1) \wedge (\neg to_1^+) \tag{11}$$

In this fashion we can continue to refine our plant model until we feel that we have something of the appropriate fidelity.

[9] This can be defined recursively through:

$$EQ_1(x_1) := x_1$$
$$EQ_1(x_1, \ldots, x_{n-1}, x_n) := (\neg x_n \wedge EQ_1(x_1, \ldots, x_{n-1})) \vee (x_n \wedge EQ_0(x_1, \ldots, x_{n-1}))$$

where $EQ_0(x_1, \ldots, x_k) := \wedge_{i=1}^{k} \neg x_i$.

4.3 Specification

As mentioned earlier the supplied specification is not usable for this model since it deals with the change of control mode whereas the supplied controller deals with the behavior in the extension mode only. However we can certainly do some reasonable consistency checks of the controller in this mode. Below we just try to make sure that the controller do not output any conflicting actuator commands:

- Never simultaneously give commands *extend gears* (u_2) and *retract gears* (u_3).

$$S_1(z) := \text{AG}\neg(u_2 \wedge u_3) \tag{12}$$

- Never simultaneously give commands *open doors* u_4 and *close doors* u_5.

$$S_2(z) := \text{AG}\neg(u_4 \wedge u_5) \tag{13}$$

4.4 Analysis

In the verification phase we try to make sure that a model $M_i(z, z^+)$ of the system *satisfies* some specification $S_j(z)$. The model is the composition of the controller $C(z, z^+)$ and the plant $P_i(z, z^+)$, i.e.

$$M_i(z, z^+) := C(z, z^+) \wedge P_i(z, z^+) \tag{14}$$

Performing the verification as outlined in section 2 all our models $M_1, \ldots, M_4$ will fail w.r.t. the specification S_1. Considering the most strict case, i.e. with the most constrained plant model M_4, we can get the controller to violate the specification in only one step through the input:

$$w_1 = 0,\ w_2 = 0,\ w_3 = 0,\ w_4 = 0,\ w_5 = 0,\ w_6 = 0,\ w_7 = 1,\ w_8 = 0,\ w_9 = 0,\ w_{10} = 0,\ \nu_1 = 1,\ \nu_2 = 0,\ \nu_3 = 0,\ to_1 = 0$$

where the signals w_i are derived from the real inputs s_j according to the source document [9]. Since the failure above is in a single step of the model it cannot depend on the fact that we have a static model of the plant. Furthermore we would have discovered this using only static analysis as well. This analysis indicates a potential error in the controller, however checking the more complete model in section 5 we see that this error was only in the partial documentation and not present in the real implementation. This illustrates one of the pitfalls of building incomplete models.

This case study could be extended with a more complete controller model (covering all modes) as well as more detailed plant model that is suitable for our needs. Finally we could also add a more complete specification. In this way we can build a more detailed formal verification of the landing gear subsystem by incrementally adding refinements to plant, controller and specification. If we find any discrepancies between model and specification we should generate a

sequence of signals that generates this error. We will then have to make sure whether or not this sequence of signals is possible in the actual implementation or not since the error could conceivably be in the documentation and not in the implementation.

5 Modeling and Analysis Based on Implemented Code

This section describes modeling and verification based on the implemented code of the landing gear controller. The purpose is to show that it is possible to perform formal verification for a system of industrial size. We work with the implemented code because we need a representation of the controller which is in exact correspondence with the implementation to ensure reliable verification results. In this case the only available representation of the implemented system is the code itself.

This section describes work which is still in progress. All results must therefore be regarded as preliminary.

5.1 Modeling the Controller

As will be shown below we perform the verification by first translating the code to Boolean algebra, which is efficiently represented by Binary Decision Diagrams (see subsection 2.3). We then analyze the code statically and dynamically by using tools that operates on the Boolean expressions.

5.1.1 The Landing Gear Controller Code

The landing gear controller on JAS 39 is implemented in Pascal86 (a Intel version of Pascal). It consists of a module of code which is mainly self-contained but in some aspects communicates with the rest of the system code. The controller code consists of approximately 1500 lines, of which about 300 handles alarm functions and pilot information.

5.1.2 Restrictions in the Modeling

The modeling has been done using a restricted class of Pascal86. The allowed data types are integer and boolean. The integer range used is $\{0, \ldots, 15\}$, 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 6. 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) and `OTHERWISE` replaced by explicit arguments.

Comment	Syntax	Domains
Arithmetic expr.	+	$\mathbb{I}^2 \to \mathbb{I}$
	-	$\mathbb{I}^2 \to \mathbb{I}$
Relation 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 6: Allowed Pascal primitives. $\mathbb{I}$ and $\mathbb{B}$ stands for integer and Boolean respectively.

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.

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.

5.1.3 Automatic Translation from Pascal to Polynomial Relations

The translation from Pascal to a polynomial representation of the code is performed in two steps. First we parse the Pascal code to an intermediate representation in Mathematica, which we call MPascal. This Mathematica representation of the code is then automatically compiled into a representation in Boolean algebra. The parsing is in this case straightforward and has been done by hand, but could of course be automated. MPascal represents the restriction of Pascal86 mentioned above.

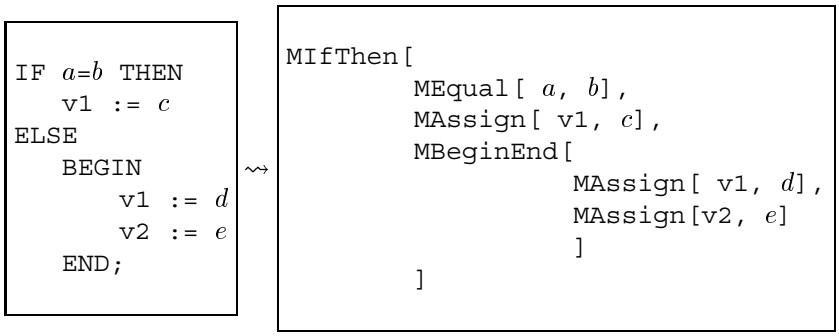


Figure 3: Parsing of an IF-THENELSE statement.

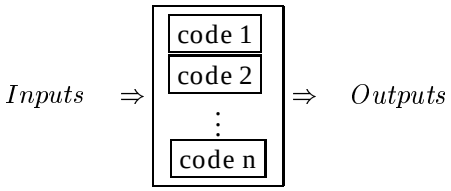


Figure 4: Part of the code, sequenced according to the control flow graph.

The parsing step will be explained further using an example. An IF-THEN-ELSE statement will be parsed to MPascal according to figure 3, where a, b, c, d and e can be any Pascal expression.

The second step in the translation produces a Boolean relation, $C(z, z^+)$, from MPascal. The program can be viewed as a sequence of code parts that, in turn, are sequences of code following the control flow graph (see figure 4). Each part of the code is essentially a function which computes and assigns values to output variables depending on input variables.

The compilation of MPascal to relations can at least be made in two different ways. One way is to first translate all primitives (the smallest parts) of the code into relations, and then combine these into larger relations. This would be a straightforward method which is easy to implement recursively. It is also easy to prove its correctness theoretically. The disadvantage is that the combination of relations require a lot of substitutions and quantifications which are time consuming operations in the present implementation of our tools. The method is therefore probably not suitable at this stage. However it would be interesting to investigate it later on in this project.

We have instead chosen to view the translation from the data perspective. If we know what variables are input and output variables, it is possible to go through the code, statement by statement, following the execution sequence[10]

[10] This is essentially the same as traversing a corresponding control flow graph.

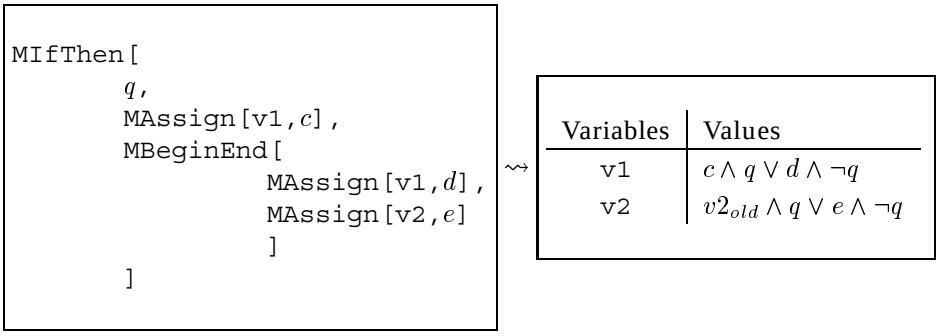


Figure 5: Compilation of an `IF-THEN-ELSE` statement.

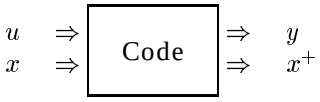


Figure 6: Signal interaction with the code.

of the code, storing information of how the values of the output variables are affected by the code and by the input variables. Only the values of the output variables are stored as Boolean expressions in each step. These expressions correspond to a data flow analysis for each variable. At the end all these expressions together with the output variable symbols are combined into a final relation, which will be used for analysis. This approach uses a kind of exhaustive simulation which in general would not be applicable. However it is usable here because of the structure of the controller code.

In figure 5 we see how the output variables in the `IF-THEN-ELSE` statement would be represented as Boolean expressions. All variables are boolean and $v2_{old}$ defines the value of the variable `v2` before this code was executed. If `v2` has not been assigned a value before, $v2_{old}$ will be equal to the symbol $\perp$. This symbol is used to indicate if there exist values of the input variables for which the value of some output variable is undefined.

Finally all variables are connected to their values and combined into a single relation. In our example above we get

$$R(z, z^+) = (v1 \leftrightarrow (c \wedge q \vee d \wedge \neg q)) \wedge (v2 \leftrightarrow (\perp \wedge q \vee e \wedge \neg q)) \qquad (15)$$

The landing gear code is one part of the software loop in the aircraft system. This means that the state of the code is stored until the next iteration of the code. If we want to write the system as in (2) we must determine which variables correspond to system state, next state, input and output. The equations in (2) can be represented by a block diagram as in figure 6, where u, y, x are vectors for input, output and state variables respectively. If we compare figure 4 and 6, we

Variable Name	Short Name	Domain	Description
utf_pagar	x_1	$\mathbb{F}_2$	Extension maneuver
tid_latt_lte_block_tid_latt	x_2	$\mathbb{F}_2$	Timer condition
spin_start_lte_spin_tid	x_3	$\mathbb{F}_2$	Timer condition
utf_start_lte_block_tid_inf	x_4	$\mathbb{F}_2$	Timer condition
utf_start_gt_luck_oppn_tid	x_5	$\mathbb{F}_2$	Timer condition
init_flag	x_6	$\mathbb{F}_2$	Initialization flag
utf_mod	x_7	$\mathbb{F}_{11}$	Extension mode
inf_mod	x_8	$\mathbb{F}_8$	Retraction mode
man_akt_mod	x_9	$\mathbb{F}_6$	Maneuver action mode
fj_mod	x_{10}	$\mathbb{F}_3$	Take-off mode
man_komm_mod	x_{11}	$\mathbb{F}_8$	Maneuver command mode

Table 7: State variables in the code.

find that $x = Inputs \cap Outputs$. In words we say that if there is an input variable that is reassigned in the code, it should be considered as a state variable. The state variables in our model of the controller are described in table 7.

Temporary variables which are neither input nor output variables can be omitted in a model like in figure 6, since we are only interested in the output behavior of the system. Still, the compiler must deal with the temporary variables in the code using them to compute the relations between input and output variables.

In MPascal we support a special form of variable declaration which not only defines the data type for each variable, but also defines if the variable is an input, output, state or temporary variable. The compiler initiates all input variables with symbols that represent an arbitrary input value. All other variables are initiated with $\perp$.

In our case we let integer variables take values in the range $\{0, \ldots, 15\}$, but for most of the integer input variables this range is too large. It is therefore important to reduce the range to avoid false input cases. Let $\Lambda(u)$ denote a Boolean expression that includes all range conditions for all input variables. Then we can compute the final BDD relation as

$$\tilde{C}(z, z^+) = C(z, z^+) \wedge \Lambda(u) \wedge \Lambda(u^+) \tag{16}$$

The valid range for each integer variable is specified in MPascal and the Λ-expression is automatically computed by the compiler.

5.1.4 The Controller Model

Let $\tilde{C}(z, z^+)$ denote the relation representing the output and state variables in the controller code. This relation has approximately 320 000 nodes when represented as a BDD. It contains 105 binary variables, of which 26 are state variables. The time required for the compilation is approximately 35 minutes on a SPARCstation 10.

In order to make the BDD representation efficient it is critical to choose a good variable ordering. The basic rule is to rank the inputs lowest and the outputs highest with temporary variables in between. This is intuitive since the output variables depend on both temporary and input variables and should be found higher up in the tree structure. Also the ordering among the variables in each of the three groups affect the size of the BDD relation. As a help in choosing ordering we have studied how the BDD grows when combining the variables. A big increase in the size when adding a variable to the relation suggests that this variable should have been ordered higher.

To improve the efficiency it would be of interest to introduce an automatic choice of variable ordering in the parsing of the code.

5.2 Analysis

5.2.1 Computation of Reachable States

In the Pascal code the controller is initialized. Using this initialization and the system relation $\tilde{C}(z, z^+)$ it is possible to compute all reachable states of the controller. It turns out that there are 10 015 reachable states and that they are all reachable in no more than five iterations of the code.

By combining the system relation with the relation describing all reachable states it is possible to further reduce the size of our model. If $R_{states}(x)$ denotes the reachable states, the reduced model is computed as

$$C_{red}(z, z^+) = \tilde{C}(z, z^+) \wedge R_{states}(x) \wedge R_{states}(x^+) \tag{17}$$

Using this reduced model it is possible to draw conclusions about the static and dynamic behavior of the system.

5.2.2 Analysis with Respect to Static Properties

By static analysis (see subsection 2.2) it is for example possible to ask questions of the type

> "Is it possible to simultaneously give the hydraulic commands extend gears and retract gears?"

The answer to such a question can be `TRUE` (or `FALSE`) in which case the statement holds (is false) for all combinations of the input signals and states. There is also a possibility that we get a relation as the answer. This relation represents all input combinations for which the statement holds.

Using our model of the controller for verification of the statement above we get the answer FALSE. This result can also trivially be deduced from the original Pascal code.

In order to try our machinery on a set of more interesting (and realistic) questions we have used material from an earlier verification project within Saab Aircraft. In this project Saab verified certain static properties of the landing gear controller with the use of NPCircuit [8], a commercial tool which essentially is a Boolean equation solver.

In these tests Saab assumed that the hardware (switches etc.) works as intended and this gives additional restrictions on the input variables. These restrictions, which we also will use, are as follows:

1. Both switches on the extension lever in the cockpit have the same value.

$$p_1 \leftrightarrow p_2$$

2. Emergency extension is not activated.

$$\neg m_1$$

3. All feedback switches on gears and doors have two contacts. These have the same value.

$$s_j^1 \leftrightarrow s_j^2, \quad j = 1, \ldots, 15$$

4. The switches on the doors does not give unreasonable values (open and closed at the same time).

$$\neg(s_4^i \wedge s_5^i) \wedge \neg(s_9^i \wedge s_{10}^i) \wedge \neg(s_{14}^i \wedge s_{15}^i), \quad i = 1, 2$$

5. The switches on the gears does not give unreasonable values (retracted and extended at the same time).

$$\neg(s_1^i \wedge s_2^i) \wedge \neg(s_6^i \wedge s_7^i) \wedge \neg(s_{11}^i \wedge s_{12}^i), \quad i = 1, 2$$

6. A gear with weight on is extended.

$$(s_3^i \rightarrow s_2^i) \wedge (s_8^i \rightarrow s_7^i) \wedge (s_{13}^i \rightarrow s_{12}^i), \quad i = 1, 2$$

7. Power supply is working.

$$\neg m_5$$

8. The variables m_3 and m_4 have values according to their definitions.

$$\begin{aligned} (m_3 = 1) &\leftrightarrow (s_3^1 \wedge s_8^1 \wedge s_{13}^1) \\ (m_3 = 0) &\rightarrow (\neg s_3^1 \wedge s_8^1 \wedge s_{13}^1) \\ (m_4 = 0) &\rightarrow (m_3 = 1) \end{aligned}$$

These restrictions can be regarded as a first model of how the landing gear works and they can be combined into one relation, which we denote $P_1(u)$.

Static analysis I We want to verify the statement

> "If extension of gears is performed (a_3) then opening of doors is performed (a_5) simultaneously."

under the following conditions:

- Hardware operates correctly in accordance with items 1-8 above.
- Nose door is closed, i.e. $(s_4^1 \vee s_4^2)$.
- Nose gear is retracted, i.e. $(s_1^1 \vee s_1^2)$.
- Previous gear command was not "extend gears", i.e.

$$(x_7 = 0 \vee x_7 = 1 \vee x_7 = 2)$$

In terms of Boolean algebra the statement can be expressed as

$$(C_{red}(z, z^+) \wedge P_1(u) \wedge P_1(u^+) \wedge (s_4^1 \vee s_4^2) \wedge (s_1^1 \vee s_1^2) \wedge (x_7 = 0 \vee x_7 = 1 \vee x_7 = 2)) \rightarrow (a_3 \rightarrow a_5) \quad (18)$$

The result of the analysis is TRUE, which means that the statement is true for any combination of inputs and states. This corresponds to the result obtained earlier by Saab.

Static analysis II We want to verify the statement

> "Retraction of gears is not performed ($\neg a_2$)."

under the following conditions:

- Hardware operates correctly in accordance with items 1-8 above.
- Nose door is closed, i.e. $(s_4^1 \vee s_4^2)$.
- Nose gear is extended, i.e. $(s_2^1 \vee s_2^2)$.
- Previous gear command was not "retract gears", i.e.

$$\neg(x_8 = 4) \wedge \neg(x_8 = 5)$$

- In the previous iteration the nose gear was not retracted, i.e.

$$\neg(x_8 = 7)$$

In terms of Boolean algebra this is expressed as

$$(R_{red}(z, z^+) \wedge P_1(u) \wedge P_1(u^+) \wedge (s_4^1 \vee s_4^2) \wedge (s_2^1 \vee s_2^2) \wedge \neg(x_8 = 4) \wedge \neg(x_8 = 5) \wedge \neg(x_8 = 7)) \rightarrow \neg a_2 \quad (19)$$

The result of the analysis is a BDD relation, with approximately 50 000 nodes, describing for which combinations of inputs and states the statement holds. We have not analyzed this result further. However it could be in correspondence with the verification performed by Saab, since they also found that there were cases when the statement did not hold.

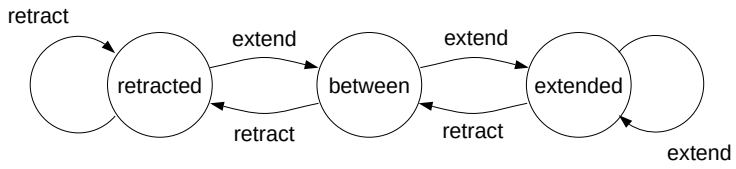


Figure 7: Gears modeled as an automaton.

5.2.3 Analysis with Respect to Dynamic Properties

By dynamic analysis (see subsection 2.2) we take the dynamic behavior of the controller into account. This means that we can verify statements that for example say that certain events never (or always) will take place. The specifications are given in terms of temporal logic.

As in the case with static analysis the answer will be either `TRUE`, `FALSE` or a relation describing all input and state combinations for which the statement holds. If we try to verify behaviors of the controller with the input signals totally uncorrelated with the actions of the controller there will be a number of "false" input-state-combinations in the answer that will be hard to interpret.

In order to reduce the number of irrelevant answers, we build a discrete model of the landing gear itself. The gears are modeled by a discrete finite automaton, see figure 7. A similar automaton is used to model the doors. These automata can be expressed as a boolean relation, $P_2(z, z^+)$. By combining the plant and the controller we get a somewhat more sophisticated model to perform verification on. Let $M(z, z^+)$ denote this model of the closed loop system. We then have

$$M(z, z^+) = C_{red}(z, z^+) \wedge P_1(u) \wedge P_1(u^+) \wedge P_2(z, z^+) \tag{20}$$

Without detailed knowledge of the desired behavior of the system it is difficult to formulate relevant specifications concerning the dynamic behavior. We are currently in the process of completing the model of the closed loop system and finding relevant specifications. No specific results on dynamic properties have therefore been achieved yet.

6 Conclusion

We have given an example of how one may verify a discrete dynamic control system by building a model of the whole process:

$$M(z, z^+) := C(z, z^+) \wedge P(z, z^+)$$

where $C(z, z^+)$ is the controller and $P(z, z^+)$ is the plant. Furthermore $M(z, z^+)$ is a polynomial over a finite field, which in this work has been the Boolean field.

We can also build a simple model of the specification using temporal algebra:

$$S(z)$$

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.

Below we briefly list some of the overall conclusions regarding this application of symbolic algebraic discrete systems theory to an industrial scale problem.

- The developed methods and tools does 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.

7 Acknowledgment

This work was supported by the Swedish National Board for Industrial and Technical Development (NUTEK), which is gratefully acknowledged.

We also would like to thank our supervisor, professor Lennart Ljung, for the invaluable support he has provided. It was he who initiated this field as part of the research agenda at the Automatic Control group.

We would also like to thank Göran Backlund (Saab Military Aircraft) for making all the effort in clearing military red tape, Ulrik Pettersson (formerly Saab Military Aircraft) who made the actual design which we have worked with and also made the partial documentation available to us and Ove Åkerlund (Saab Military Aircraft) who assisted us with reasonable static analysis questions.

Finally we would like to thank the other COHSY participants for stimulating discussions.

References

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

[2] CCITT. *CCITT Recommendation Z.100: Specification and Description Language SDL*. CCITT, ITU General Secretariat – Sales Section, Places des Nations, CH-1211 Geneva 20, Switzerland, 1988. Blue Book, Volumes X.1–X.5.

[3] CEI-IEC. Preparation of function charts for control systems. Standard 848, IEC, 1988. First edition.

[4] 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.

[5] Roger Germundsson, Johan Gunnarsson, Arne Jansson, Petter Krus, Magnus Morin, Simin Nadjm-Tehrani, Jonas Plantin, Magnus Sethson, and Jan-Erik Stromberg. Complex hybrid systems I: A study of available tools and specification of planned work. Technical Report LiTH-IDA-R-94-29, 1994.

[6] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[7] IEEE. *The IEEE Standard VHDL Language Reference Manual*. IEEE, 1987. The IEEE 1076–1987 standard.

[8] Logikkonsult NP AB. Using NP-circuit to model and verify full-scale systems. Preprint, 1994.

[9] Ulrik Pettersson. Ll-oh, 1992. Slides and other documents used in a presentation of the landing gear process.

[10] S. Wolfram. *Mathematica. A System for Doing Mathematics by Computer*. Addison-Wesley, second edition, 1991. ISBN 0-201-51502-4.