Abstract

There was an increment of attention into Model Predictive Control (MPC) in the industry. Nowadays, it is a well known engineering implementation in the academic sector and industry. Despite the fact that MPC has evolved in various areas, there is not a tangible procedure to set up a conventional V&V routine, given its functionality.
Consequently, MPC's validation and verification (V&V) of its implementation in certain industries has been an impediment for development of this control strategy. As a case in point, in the avionics industry the V&V of any system is crucial for its certification proceedings and approval.

In this study, we research on the predictive control algorithm and explore ways to verify the functionality of the control system. The project has two main components, the first is the introduction of the MPC, theory involved and its block model described in Simulink. Verification at this stage was set up for a online-verification (verification at runtime simulation). Afterwards, this block model generated C code for its implementation.
The second component of this project was to verifying the model translation in C code, and check its consistency. Here we built a C project with the code generated. Verification at this stage dealt firstly with tracing assertions from the block model to source code. Then by using a formal verification tool, we tried to find uncovered “bugs” in the system. An introduction of bounded mode checked is also included.
The verification and validation of this complex control system produced a reports in coverage at both stages. In the block model(Simulink) the Verification and Validation tools was used. On the other stage GCOV and LCOV were used for the code generated.
Additionally, the bounded model checked CBMC provided us with a report on counterexamples that are crucial for bug-hunting during the V&V routine.

The project indeed showed us an innovative way to employ robust formal verification techniques to develop a more flexible V&V design for a MPC that was able to trace assertion along a hybrid verification design test.

Contents

1 – Introduction

Since the late 1970's there was an increment of attention into Model Predictive Control (MPC) in the industry. Its advantages against usual control systems (proportional integral derivative -PID) are many. MPC employs a direct-defined model for prediction of the system's outcome. This “forecast” represents the output at future time instants called horizon. And continuously this horizon is shifted forward in future. This approach provides efficiency in the control of the system and long operability time without much intervention. However, there is not a tangible procedure to set up a conventional V&V routine, given its functionality. This control system technique can be easily implemented, but its derivation may be complex. Furthermore, if the model involves a dynamic process and constraints, the results will be an increment of computation per sampling-time and per-constraint respectively.
The validation and verification of its implementation in certain industries has been an impediment for development of this control system (e.g. Avionics industry). This project will expose a new way to tackle this problem by the utilization of V&V techniques used in the microelectronic industry.
In this section we will explain the objectives of the project. These objectives are milestones along a procedure that now can be used to set up a verification design for a predictive control. Additionally, this section will inform about the structure of this report.

1.1.- Aim and Objectives

The objective of this project is to analyse an autonomous robotic system controlled with MPC strategy. Being this a navigation vehicle, the preliminary study will focus in the algorithm exerted. From a basic to a more detailed perspective, the controlled systems technique will be broken down where possible for further analysis. There will be a special considerations towards the constraints used in the model of the control system mechanism. Altogether, a process of identification for potential verification methods will be carried out. On that account, this project aims to focus on exploration of methods, tools and implementations able to satisfy relevant standards. Depending on the feasibility and practicality of the plant model conception, V&V process may be taken at different levels. As a reference point for validation, the RTCA/DO-254 guidance for hardware for Airborne[9] will be used. Additionally, the software and verification of its functionality is of importance[8], given that the cost of maintenance afterwards a certification of an inadequate software package would be expensive, even more if we talk about safety-critical software as it is used in avionics.

1.1.1.- Verification for MPC Block Model

Initially, the study will investigate the basic of control system strategy -MPC- in use. This will be done in order to understand the complexity that the controlled model may reach. This will allow a better approach in terms of specifications needed to be accomplished for the verification process. The algorithm is implemented in Simulink[11], and it will be studied, trying to obtain boundaries and properties required for a test-bench. This study is compulsory, since the complexity of a MPC system may be increased during the online optimization, incurring in more computational effort or a non-deterministic algorithm[1]. This will make the verification assignment unattainable. Once specifications are determined, the MPC block model will be checked for correctness. At this stage, a runtime-verification will be developed. The constraints in the block model will be correlated to the coverage report. In this phase, the Validation and Verification Toolbox will be used in Simulink. Here it will be exploited the benefits of synchronous data flow used in Simulink[2].

1.1.2.- Verification for C source files of MPC model

After having the systems specification and properties clearly stated, a study of the tools available will be carried out. In contemplation of these results, an appropriate method was chosen. Where possible, more than one verification technique/method will be employed in order to expose the trade off between these approaches. This was done in order to select the most suitable tool for the analysis of the generated C code. It was important to investigate how to build a C type project, compile it and corroborate the executable's

output (*.mat files). Afterwards, it was a must to analyse the generated source code to see how and where are assertions were placed. Furthermore for our particular study, the assertions had to allow traceability from the block model to the source code. This was of great importance since it represents the key to make feasible the usage of a bounded model checker.

1.2.- Structure

This report consists of 7 sections, where firstly time is spent explaining the background needed to understand the basics of model predictive controllers. In the same section it is included information about the formal verification tool employed in this hybrid verification procedure. This include the basic definition of a model checker, SAT solver and ends mentioning the capabilities of the tool CBMC. The following section focused on the description of the initial MPC design. It describes the block-model used to set up a predictive controller in Simulink and how this is presented in the autonomous vehicle. This section include V&V factors to be considered in the software development for safety-critical packages and tools to be used in our DUV(device under verification). The third section concentrates in describing how a test-bench is set up by using the tools available in Simulink (basic block construction) and the theoretical background on MPC to establish assertion. This section worked on the assertion-based test-bench and explains the V&V tool used for analysis of coverage. The next section deals with the task of building a C solution in Microsoft Visual Studio with the code obtained by the code generation toolbox from Simulink. It also explains how the translation was check for correct execution of the model. The fifth section is centred in the traceability of the property blocks constructed in the high-level model simulation. In this section, it is explained how the assertion set for the physical constraints and the optimizer of the MPC are analysed with the bounded model checker CBMC. Right after that, the next section covers the results obtained from the previous testing. This includes the coverage for the high-level mode and also the coverage for the C source code which represents the lower-level description of the model. Finally, in the discussion section it is explained how and why this method was developed and it contains a evaluation subsection that summarises that aims and objective achieved in this project and ends by presenting factors that could have brought additional development in the assembling process of the verification environment. This is added in the future work subsection that explains the new approach exposed in this research and implementation work plan.

2 – Background

2.1.- Model predictive control.

Model predictive control is considered as an advanced control method that has influenced significantly on the industry. The main reasons why this strategy is well accepted are many. It naturally controls multivariable systems while taking account of actuators limitations (prediction is made using data of system previous states). MPC allows operations close to constraints and its control update rates are relatively low. In the last decade, in the computer world the hardware speed has increased by a factor of 10^6 [RTV97] and optimization algorithms are being exploited in various engineering fields. This has led to an increase of usage of predictive control systems. The control strategy used in model predictive control employs an algorithms that has a explicit process model. This Model predicts the future output response of the plant. Optimization is used at each control interval. The aim of the optimization is to reduce the difference between the desired (reference trajectory) and the actual responses, by means of a cost function optimizer. The first element of the optimal sequence (vector) is applied to the plant and the whole calculation is repeated in the following control interval.

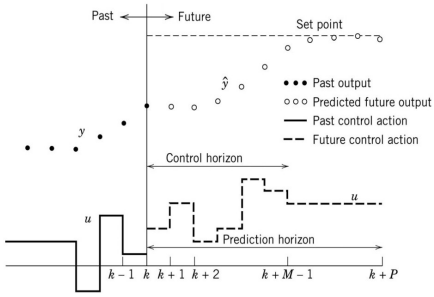


Fig.1. Predictive control

2.1.1.- Constraint awareness in predictive system

It is clear that operability of a plant (model) should not be running at its limits within boundaries. But the control systems ideally should be able to deal with disturbances from various sources that were not expected. Hence, optimal control system, that deals with these disturbances, is able to operate closer to constraints. In linear optimal control, it is considered that disturbances are random and if it is possible to reduce the variance of the outputs the control systems may be able to operate as close as desired to the constraints. Consequently, the plant would be operating at a more optimal performance.
To illustrate this, in Figure 2 we can see probability distribution of three cases for controlled output of a plant. The red vertical dashed line is the constraint that should not be violated. The distribution (a) has a typical Gaussian shape with a high variance. This is caused by the ineffective tuning of a linear controller. So for us to have a plant operating with a low probability in violation of a constraint, the set-point has to be set far from the constraint. In this case the plant would be operating far from the optimal performance for the majority of time. (b) the second curve represents a distribution obtained by an optimized control system (it may have been linear). It is clear that this second has a reduced variance, letting us place the set point nearer to the constraints. The distribution still keeps Gaussian form because of the optimizer linearity. The third curve, (c) shows how a predictive system controls the probability distribution of the outputs of the plant in a different way. The model predictive control is aware of the constraints. Consequently, it responses according to the disturbances. The MPC will react in one way if the disturbance's effect push away the output(s) from constraints and in a different way if this effect pulls output(s) toward the constraint. Something notable here is that the outcome of the MPC probability distribution of plant's output is not longer described as a Gaussian. This is due to the non-linearity of the controller. In the Figure2, it is clear that curve (c) is not symmetrical, but still keeps an acceptable small probability of violating a constrain.

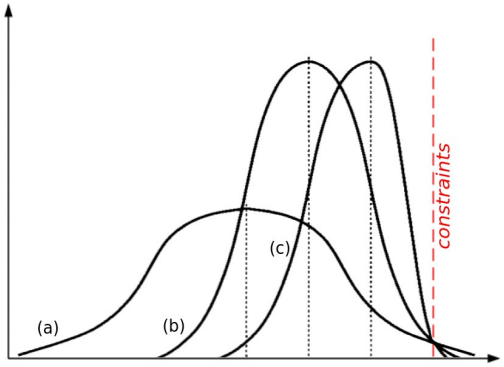


Fig.2. Set-point improvement from the use of predictive control [10].

2.1.2.- MPC basic idea

To understand how MPC works, we will describe mathematically a simple SISO (single input,single output) example. Let us assume the following variables:

k : time step
y(k) : current output
s(t): set-point trajectory, represents the ideal output path that should be followed.
T_s : sampling interval
ϵ (k)= s(k)-y(k) : error at time step k.
r(t|k) : reference trajectory, runs from current output y(k), toward s(t) describing the ideal path in which the plant reaches its set-point.

Additionally, we understand that the reference trajectory links the the current output y(k) to the ideal output s(t). Usually, y(k) approaches to s(t) exponentially having a time constant T_{ref} which represents the system's responsiveness. r(t|k) is of great importance since it describes important aspects of closed loop behaviour of controlled plants.
The reference trajectory is selected in order to have it defined in i steps later like:

$$\epsilon(k+i)=e^{-iT_s/T_{ref}}\epsilon(k)$$
$$\epsilon(k+i)=\lambda^i\epsilon(k);0<\lambda<1$$

And we define the reference trajectory as follows:

$$r(k+i|k)=s(k+i)-\epsilon(k+i)$$
$$r(k+i|k)=s(k+i)-e^{-iT_s/T_{ref}}\epsilon(k)$$

Afterwards, as we know the predictive controller has a internal model that is employed to periodically predict the output. It does it at current time over a future prediction horizon. This behaviour depends on the input trajectory û.

$$\hat{u}(k+i|k);i=0,1,2,\ldots,Hp-1$$

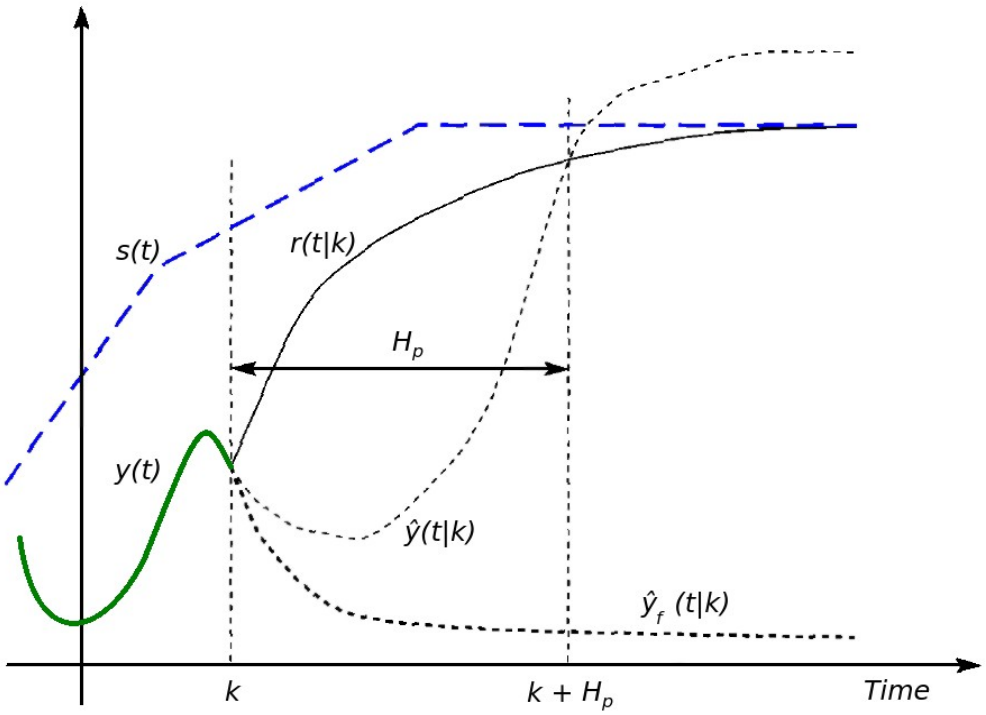


Fig.3.Model Predictive Control Strategy[20]

The aim here is to chose the input which promises the best predicated behaviour for the plant. In this let us assume that the internal model is linear, and consider we have the output measurements y(k) available at time of decision when selecting value u(k).
The internal model must be proper, meaning that the y(k) does depend on past inputs. There may be several input trajectories{û(k|k) ,û(k+1|k), û(k+2|k), û(k+3|k), …, û(k+H_p-1| k)}, and we select the input trajectory that brings the plant output at the end of the prediction horizon (at k+H_p) to the desired value of r(k+H_p). In this case we have a single coincidence point.
For simplicity, we assume that input remains constant over the prediction horizon, that is

$$\hat{u}(k|k)=\hat{u}(k+1|k)=\hat{u}(k+2|k)=\hat{u}(k+3|k)=,\ldots,=\hat{u}(k+Hp-1|k)$$

Consequently, we will have one parameter û(k|k), and remembering that the equation required for the predictive model asked us to satisfy the following

$$\hat{y}(k+Hp|k)=r(k+Hp|k)$$

This clearly shows that we nee to solve a single equation. Once a predicted trajectory has been selected, we use the first element and we apply this into the input signal of the plant. u(k) is the actual signal applied.

$$u(k)=\hat{u}(k|k)$$

Afterwards, the complete cycle (output measurement predictions and input trajectory determination) is repeated in the following sampling interval. Later a new output will be measured, and a new reference trajectory is defined

$$y(k+1) \quad \text{and} \quad r(k+i|k+1);i=1,2,3,\ldots$$

Predictions will be made over the horizon $(k+i+1);i=1,2,3,\ldots,H_p$

And the new input trajectory is selected as follows:

$$\hat{u}(k+1+i|k+1);i=1,2,3,\ldots,H_p-1$$

Then the next input is applied to the plant, the input will have the form of

$$u(k+1)=\hat{u}(k+1|k+1)$$

In this simplified example, the prediction horizon remains of the same length as the preceding operations. However, it slides along toward the future in one sampling interval at each step. The technique of plant control is called receding horizon.

2.1.3.- Optimal input computation

Usually in a MPC there are several coincidence points, that is all the points k+1, k+2, …, k+H_p represent coincidence point in the trajectories. Despite the fact that there are more parameters to choose from (more options in the selection of input vectors), there are more coincidence points than parameters. This will merely result in unsolvable equation system. Too many equations to be satisfied with unavailable variables. At this point is where optimization algorithms are used. It is clear that we need to approximate and for this, it is common to use a least-squares, that sums up the square of error and it's minimized:

$$\sum_{(i\in P)}[r(k+i|k)-\hat{y}(k+i|k)]^2$$

P represents the indices related to the coincidence points. Again considering a case with one coincidence point, [k+H_p] and one parameter to select û(k | k), then the internal model is employed to predict the free response of the plant defined as :

$$\hat{y}_f(k+H_p|k)$$

This free response would achieved the one of the coincidence point if the future input trajectory maintains its value equal to the latest one:

$$u(k-1)$$

We define S(H_p) as the response of the model to a unit step input. Then we will have "H_p" steps being applied to the model. Afterwards, as predicted output at time [k+H_p] we will see that this is given as follows:

$$\hat{y}(k+H_p|k)=\hat{y}_f(k+Hp|k)S(H_p)\Delta\hat{u}(k|k)$$

where Δû represents the change from current output u(k-1) to the predicted input û(k|k)

$$\Delta\hat{u}(k|k)=\hat{u}(k|k)-u(k|k)$$

Since we aim

$$\hat{y}(k+Hp|k)=r(k+Hp|k)$$

the optimal variation or change of inputs is described by

$$\Delta\hat{u}(k|k)=\frac{r(k+Hp|k)-\hat{y}_f(k+Hp|k)}{S(H_p)}$$

This is true assuming that plant and model output are equal until time k, which in practice does not happened. Still, this exposes how optimization needs to be analyse and recheck the algorithm because the environment where the control is applied may have different input sequences (not just a step input) and bring up unexpected predictions far from constraints and delaying the responsiveness.

Continuing the mathematical example, if we have more than one coincidence point, and limit them to "c" points, then the values of the reference trajectories are :

$$r(k+P_1|k)=r(k+P_2|k)=r(k+P_3|k)=,\ldots,=r(k+P_c|k);\ P_c\leqslant H_p$$

And in this case we will aim:

$$\hat{y}(k+P_i|k)=r(k+P_i|k);\ i=1,2,3,\ldots,c$$

and to select Δû(k|k) to resolve the equation system:

$$\begin{aligned}
\hat{r}(k+P_1|k)&=\hat{y}_f(k+P_1|k)+S(P_1)\Delta\hat{u}(k|k)\\
\hat{r}(k+P_2|k)&=\hat{y}_f(k+P_2|k)+S(P_2)\Delta\hat{u}(k|k)\\
&\vdots\\
&\vdots\\
\hat{r}(k+P_c|k)&=\hat{y}_f(k+P_c|k)+S(P_c)\Delta\hat{u}(k|k)
\end{aligned}$$

solving this equation system will provide us with the correct prediction. If we group these variables such as:

$$\tau = \begin{bmatrix} \hat{r}(k+P_1|k) \\ \hat{r}(k+P_2|k) \\ \cdot \\ \cdot \\ \cdot \\ \hat{r}(k+P_c|k) \end{bmatrix} \qquad Y_f = \begin{bmatrix} \hat{y}_f(k+P_1|k) \\ \hat{y}_f(k+P_2|k) \\ \cdot \\ \cdot \\ \cdot \\ \hat{y}_f(k+P_c|k) \end{bmatrix} \qquad S = \begin{bmatrix} S(P_1) \\ S(P_2) \\ \cdot \\ \cdot \\ \cdot \\ S(P_c) \end{bmatrix}$$

The solution could be obtained in Matlab with the operator " \ " as indicated below:

$$\Delta \hat{u}(k|k) = S \backslash [\tau - Y_f]$$

Mind you that this would solve the equation system where there are not constraints on the inputs nor outputs, and the least-square solution is acceptable. Otherwise, we need to replace the algorithm for a constrained least-square algorithm.

2.1.4.- Setting up a MPC

The main concepts to be consider for the implementation of a MPC are basically three. The first one the *model* to be employed for the predicting the output response of the plant. These models can be Finite impulse response(FIR) and step response. FIR response model provide a clear description of the process time delay, response time and gain, but only in stable plants. Step responses models. FIR and Step responses models employ dynamic matrix control[12]. Another model, Transfer function models are used and can be applied to unstable and stable plants[13]. This model does not give much description of the process and it is not effective handling multivariable models. However, it is practical for implementation due to few parameters required for the modelling of all processes. The last model and the one is getting more popularity is the state space models[2,14]. This model does manage multivariable systems.

The second one encloses data of the state of plant, meaning the *measurement*. This is a crucial and needed to the model in order to asses the current plant status at the start of every single control interval. In reality, most of these measured data is not available during processing at time '*k*'. Hence, estimation is used in order to proceed the calculation.

As a third concept to be acknowledged we have the *way of control*. Meaning how the best approach will be taken, considering the constraints and the optimization requirements. Predicting with an updated data available and within a moving horizon time span.

Along the time, MPC implementation has been changing according to the control system model to be used. These configurations started from Unconstrained infinite horizon [21], Constrained finite horizon – MPC [22], Posed as quadration program[23] to an explicit MPC[24]. All these milestones in MPC will give more detailed perception in how the algorithm can be break down into sections for their verification. In this closed loop MPC model, the automatic code generator will have to accomplish all specification requirements.

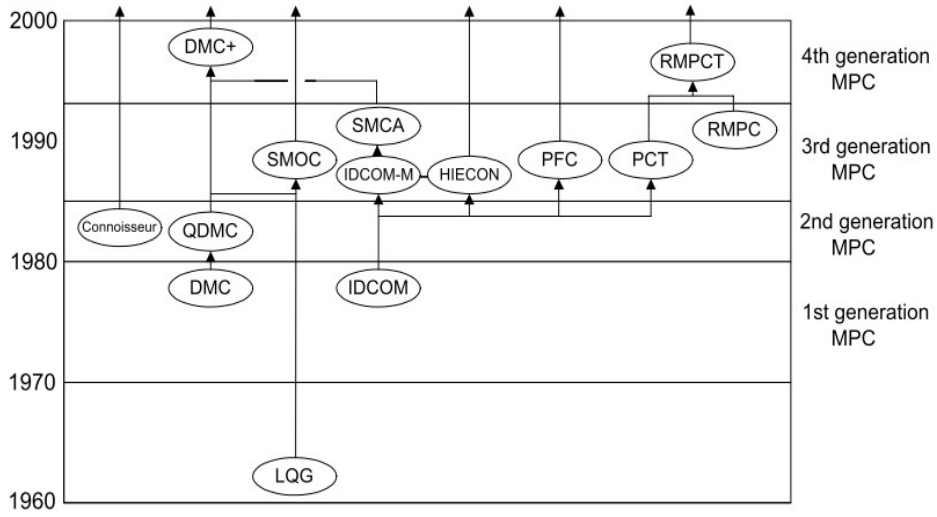


Fig.4.Genealogy of linear MPC algorithms[25].

Model predicting control evolved and its robustness in the modelling of process has seen improvements [16,17,18,19]. For our model, it will be employed predictive control using a receding horizon strategy.

2.1.5.- Constrained Receding-Horizon MPC

The basis of a continuous-time model predictive control is to minimize the cost function. This cost function is quadratic and represents the error caused by the difference between the ideal output and the actual result[1,20]. Its reduction would indicate a better performance of the model-based system control. This system can be represented as :

$$x(k+1)=Ax(k)+Bu(k)+w(k)$$

where, w(k) is the disturbance (external noise). The control system itself will have to conform with the constraints, in this case polyhedral constraints.

$$(x(k),u(k))\in C\,\forall\ k$$

The simple constrained control problem to solve in a continuous time is expressed as the minimization of quadratic cost function dependent on linear inequality constraints, which is a quadratic programming problem. The function to be minimized is:

$$J_{\infty}=\sum_{k=0}^{\infty} x(k)^T Qx(k)+u(k)^T Ru(k)+c^T x(k)+d^T u(k)$$

Hence, for an stable control strategy, it is needed to minimize J function, satisfying constraints

$$u(\tau)\in U\,,x(\tau)\in X$$

And accounting robustness

$$\sum_0 \in S$$

The General Prediction Control(GPC) model has been quite accepted in industry and academic world. But the main drawback of this was the lack specific results of the properties of predictive control and an unclear question about stability(for different plant model) and robustness (against new sources of disturbance).
There is not a clear theory that establish the relationship between design parameters and the closed loop conduction. This alludes to the weighting sequences and horizons.
Considering the previous limitations, a new formulation was developed. The Constrained Receding-Horizon Predictive Control (CRHPC) overcomes the stability issues and robustness. Hence in this project, the control algorithm will employ this strategy for a multivariable systems.

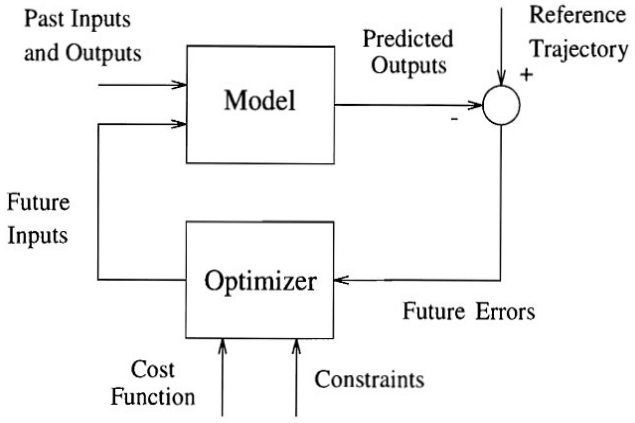


Fig.5 General Structure of MPC [20].

Furthermore, the control algorithm -which is a code generator for the of the autonomous system- is set up in a closed loop. It will involve a multivariable constrained system that works on a feedback for stability.
There will be special concern about the constrains, since these are likely to be the starting point for the V&V analysis.
In MPC, the management of constraints is one of its main qualities. Constraints can easily included in the optimization process and they can limit upper and lower limits of inputs, input rates, limitations in the output and within stages. Therefore, a combination of these constraints can also be conceived.

2.1.6.- Multivariable MPC

As it could be expected, most of the MPC are not set handling a single variable, instead they are driven in a multivariable system. Referring to these, the control of MIMO (Multiple Input Multiple Output) is how the control of the autonomous system will be treated.
Multivariable systems do have performance index J (cost function). One of the main features of MPC is that it is a systematic tools, meaning, it performs handling interaction and stabilising multivariable systems. As result, MPC provides the best predictive performance.
A poor selection of the cost J could enlarge the effects of implicit interaction in multivariable systems[27]. This would gives bigger values for J and would not be the optimal approach. Consequently, the system will have poor performance for sensibly chosen horizons. This interaction may result in loops that are faster than other and/or are tightly controlled.
To counteract these undesired effects, MPC algorithms can extend J with more specific weighting matrices. For example the cost function is often represented as:

$$J = \|r - y\|_2^2 + \lambda \|\Delta u\|_2^2 = \|e\|_2^2 \lambda \|\Delta u\|_2^2$$

And including matrix weights , lets say W_y , W_u , the previous formulation could change to :

$$J = \sum_{i=n_w}^{n_y} \|W_y(r_{k+i} - y_{k+i})\|_2^2 + \lambda \sum_{i=0}^{n_u-1} \|W_u(\Delta u_{k+i})\|_2^2$$

Here the matrix weights are positive, definite and diagonal, meaning that they can vary within the horizon *"i"*. Quite often the responsiveness of the resulting controller parameters is slow because of the weights can be unnoticeable. Hence, during the simulation of the robotic system it will be needed to test while modifying the magnitude of the weights to be able to perceive a significant adjustment.

The tuning process of these weights can be tedious, specially if it is needed stability in the tracking behaviour of each loop and their correspondent inputs.
As it can be seen, the modulation of a multivariable system involves more complexity and puts on the table a clear trade-off present in MPC deigning. Despite the fact that it is completely arbitrary, this trade-off is defined between the size of interaction and the response times of each loop. Nevertheless, it would be possible to formalize this arbitrariness by producing a mathematical template. This will increase complexity (non-deterministic algorithm), probably making it difficult to verify it. Moreover, it may result in a less responsive system.

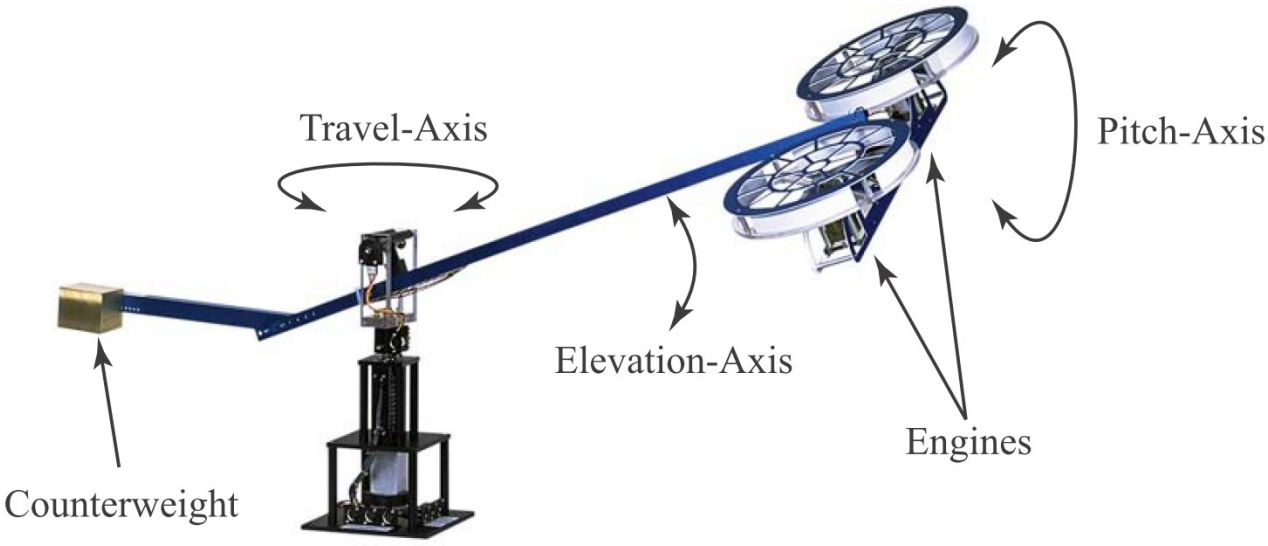


Fig.6.Multivariable example : Quanser 3-DOF Helicopter[27]

The above picture depicts a non-linear MIMO system. In a multivariable system there is great concern about the constraints that can be hard and soft. In real-life problems, it is common to have as desirable constraints working in an inconsistent manner. This means that they are not being accomplished simultaneously. This will definitively not provide a solution with a predictive control.
Consequently, there is a need to reorder these constraints and separate them into hard and soft limits. Hard constraints are defined and are meant to be satisfy at all times. Soft constraints are more flexible in a sense, giving the chance to be violated but paying a penalty, such as loss of safety-critical response (important the robotic system) and generally product/solution quality. There will be special emphasis in the handling of soft constraints so that if solution cannot be given, one of the soft constraints can be relaxed in order to reach a feasible solution. The criteria applied toward this technique of relaxing certain constraints may depend on the process.

2.1.7.- Summary of Model Predictive Control

As it has been exposed in the previous subsections, MPC is an advanced control systems that regards very much to a model-based control that acknowledge the constraints present in the input and output of the plant. Moreover, it is robust against external and unexpected disturbances that tend to decrease efficiency of an usual control (e.g. PID controller). MPC is a flexible strategy, hence there are various types of predictive controller that are commonly classified according to the plant model that is replicated internally for prediction signal. The theory covered tries to be concise in order to grip a basic understanding of a predictive controller, needed to understand the MPC block model and be able to set up some testing within the high-level model.

2.2.- Bounded Model Checker - CBMC.

This section explains the basics of model checker. It also describes a model checker tool that verifies ANSI-C programs. This tool is CBMC and there will be few examples that outline the functionality of the bounded model checker. This information will be crucial in order to understand how CBMC produces a counterexample to verify a property formerly defined in the C/C++ code.

2.2.1.- Model Checker definition

In computer science model checker is a formal verification technique that is widely used for verification and validation of software. It is a proper "bug" hunter. Model checker are commonly used in hardware designs. However, in software given a larger combination present for structures, this approach can not be completely algorithmic due to the presence of undecidability (hence usage of a bounded model checker). A model checker is indeed a tool for automatic verification of properties (set up from specifications) within a finite state system. In software engineering and microelectronic industry, model checking works with a model of a system exhaustively to ensure that the model fulfils the requirements. Quite often these hardware and software requirements are tied to safety-critical specifications. Hence, model checker focuses on the violation in a logic sense of a specification, in order to exercise a non covered part of the algorithmic/model that could bring up an unexpected and systems reaction. This indeed would be catastrophic for a system in aeronautics an also disastrous for a new hardware/software development that exposes flaws in its late stages of design/fabrication.

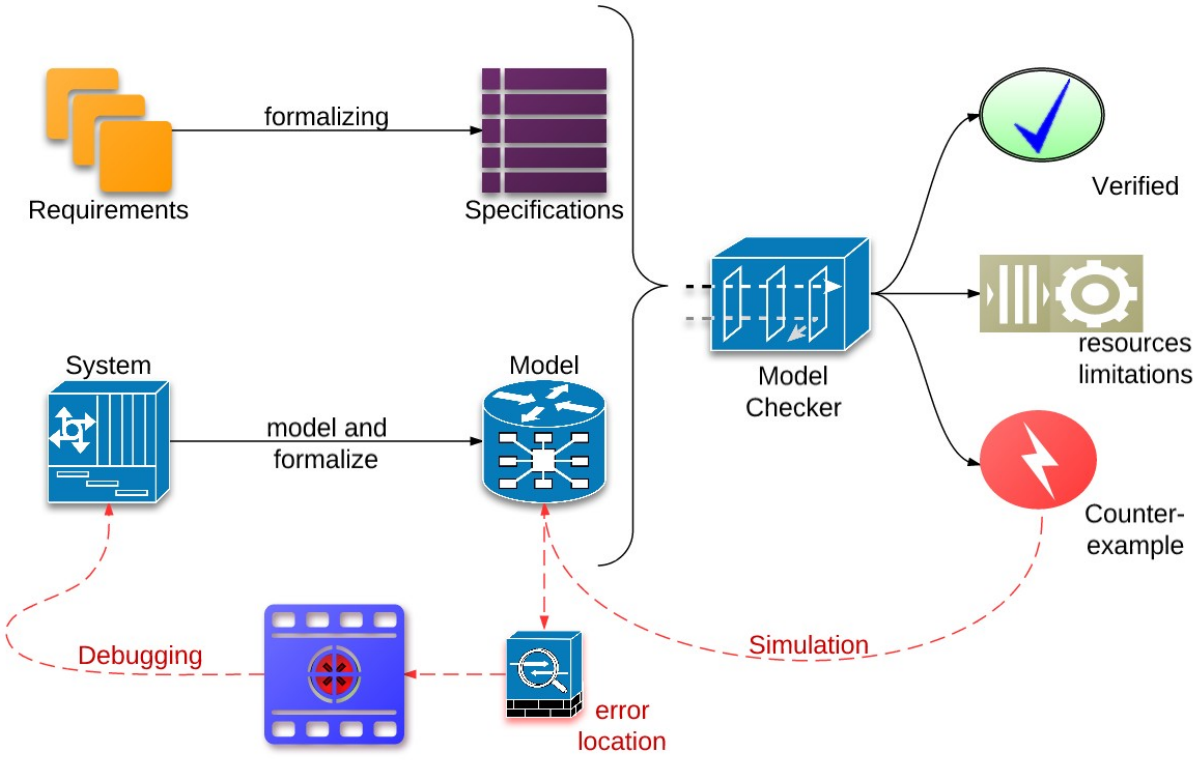


Fig.7.Model Checker strategy

Therefore, the main idea of using a automatic model checker is that having a "system" (could be a hardware or software) and a "claim" (which is represented by requirements, properties, specifications, etc.) use a SAT solver (could be used a SAT or SMT solver) to find out if there is an execution that violates the claim. In this case it will be used a SAT solver that works with CBMC. In ANSI-C programs the claims that are checked by default are dynamic memory management, pointers consistency and bit-vector arithmetic structures. Additionally, for a safety-critical system set up and coded in C/C++ the model checker has as claims simple properties, such as:

- Division by zero.
- Pointer checks (for example pointer dereference)
- Array bound checks (like buffer overflows)
- Dynamic data types
- Bit vector operations
- Arithmetic overflow
- User defined assertions
- Non-determinism

2.2.2.- SAT Solver

Commonly called boolean satisfiability, SAT is a proportional formula that determines if there exists a variable assignment so that the formula evaluates to true. SAT is likely to be the most studied of combinational optimization problems. Considerable research has been focused on solving efficiently this problem. This problem is encountered in applications ranging from Electronic Design Automation to Artificial intelligence. The boolean Satisfiability is classified as a NP-complete problem due to its role in theoretical and practical applications.

Boolean formula of SAT are often expressed as conjunctive normal form (CNF), which is composed of one logical clause and additional clauses. Each clause represents a logical literals or more than one. The term literal refers to a positive or negative form of the variable. In order to satisfy the consistency of a CNF boolean formula, each clause must be satisfy separately. The improvements of SAT solvers are is notorious and can be seen in Figure 8. SAT solvers are quite attractive because of its efficiency, automation and its support for features that can handle programming languages formats such as bitwise operators, pointer arithmetic, dynamic memory and type cast. And in the last decades it has been seen that the improvement are thanks to an efficient word-level reasoning and array decision procedures. SAT solvers aim only finite-state systems. It can not explore infinite execution length (i.e. loops).

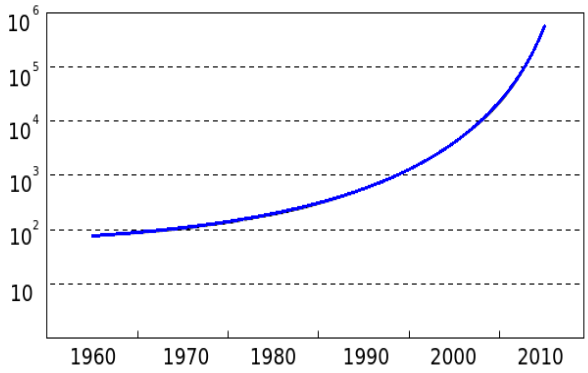


Fig.8. Number of variables of a typical SAT instance that can be solved by the best solver in that decade

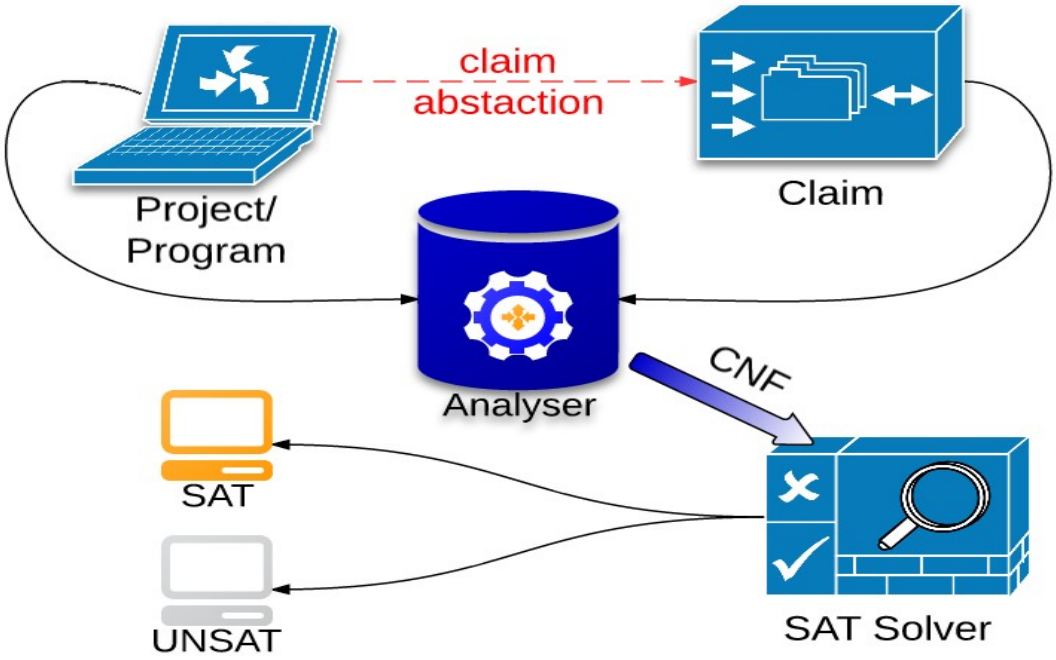


Fig.9. SAT solver abstraction. This graph describes the how Minisat2 is deployed.

2.2.3.- Bounded Model Checker (BMC)

In software systems, most of safety-critical sections are written in low level language like ANSI-C or assembly language. This is done due to the performance requirements but programs at the same time represent the main cause of safety problems that can occur. In programs written in high level language present a lower risk in these terms. Consequently, verification of a ANSI-C code is more complicated owing to the usage of pointers, arithmetic, pointer arithmetic and bit-wise operators.
BMC does implement a transition relation for a complex state machine. The specifications/requirements are unwound to obtain a boolean formula. Then by using a SAT procedure, the formula is checked for satisfiability. When the formula is not satisfiable, the checker unwinds deeper in order to find a longer counterexample. BMC unwinds sufficiently to make sure that a longer counterexample exists.

2.2.4.- CBMC

This tools that was first presented in [38] is a bounded model checker for C/C++. In the same way a compiler works, CBMC takes the .c file(s) as arguments to be declared in a command line. Afterwards, it translates the C program and combines the definitions of functions from the files, emulating a linker but this does not create a binary for execution. Instead CBMC achieves a construction of a symbolic simulation of the program.
The way CBMC works with a program is effectively formal and automatic. It first simplify the control flow of the program. Then unwinding takes place in every loop located in the code. It does a conversion of Single Static Assignment (SSA) so that equations built up. Bit-blasting is used in conjunction of a SAT solver. Finally, CBMC converts the SAT assignment into a counterexample for the equation under the scope. In order to understand how CBMC works, in the following section there will be a explanation on how this tool checks for properties violations in the different features of ANSI-C. Afterwards, it will be presented some examples that exposes the robustness of this bounded model checker.

2.2.4.1.- CBMC attributes of BMC in ANSI-C

*Data types and operators

CBMC fully supports ANSI-C boolean and integers arithmetic operators on scalar variables. Meaning that this tool will check for consistency when when using the following operators: !x, x&&y, x||y, x+y, -x, x-y, x/y, x*y, x%y, x<<y ,x>>y, x&y, x|y, x^y, x<y, x>y, x<=y, x>=y.
One very important point to be mention here is that the operations of divisions, multiplication and remainder (/,*,% respectively) represent a larger usage of resources when using a SAT solver. When CBMC does the conversion and passes it to the SAT solver, the equation is expanded dramatically. This happens because CBMC in a non deterministic way selects two unsigned variables for one of these operations (e.g. multiplication) and by using shifts and additions obtains the final result to be asserted. Despite the fact that in code appears as x*y, the SAT instance would be translated in about 2400 variables. To solve this using MiniSAT would take longer than 2 minutes.

The properties checked in this section are arithmetic overflow for most of arithmetic operations. In division and remainder operations, division by zero is checked.
CBMC also provides support for arithmetic operability of floating point. Float, double, long double are included x+y, -x, x-y, x*y, x<y, x>y, x<=y, x>=y. Again the multiplication operation will be passed to the SAT solver with many variables. This tools allows type conversion from and to integers.

* Types cast

CBMC provides support for arithmetic type cast. In this feature, this tool checks for unsigned data types. For example that no overflow exception occurs. There is an overflow exception for signed data types. Optionally, it is possible to check for this overflow as well.

* Side-effects and functions calls

CBMC recognize all side-effect operators within their respective semantics. These operators =,+=,-=, post increment and decrement are checked for consistency and the effects in the code. Besides, the exceptions of &&,|| and ?: (ternary operator), ANSI-C allows arbitrary execution ordering. CBMC knows about this and prevents that errors to be hidden during the abstraction of the boolean formula from the code. Hence this tool allows a predetermined ordering of evaluation of all the operations present in the code. Meaning that CBMC will ensure that no side-effect alters the value of variables that are evaluated within a same rank of priority. This also cover the modifications made indirectly by the usage of pointers. This bounded model checker outlines the architecture-dependent behaviour of the program in order to stablish a fixed ordering for the evaluation and consequently avoid a undecidability while abstracting the boolean formulation. It is important to keep in mind that ANSI-C programs support this dependency, it is no desired while while constructing a logical formulation.
When coding functions, CBMC provides a in-lining approach. There is not a modular course for function call in this tool. The locality of static and non static variables is kept by renaming and using additional parameters in the boolean expression. If recursion structures are presented in the code, CBMC employs a finite unwinding to reach a assertion to be translated. This tools does unwind sufficiently in order to grip a point state from where a counterexample can be found.

* Conditional statements, control flow and loops

Standard ANSI-C supports various conditional statements for programs. In the same way, CBMC allows this basic code blocks. The checker here generates warnings when the assignment operator is employed as conditional that could change the control flow (i.e. if, when). The usage of GOTO is not that popular but CBMC supports this feature. This tool knows where it is a backward or forward jump in the code. The RETURN (with no value) statement is understood as a GOTO. The one returning a value is understood as an assignment of a value returned and a GOTO statement pointing to the end of the function call. CBMC check effectively that non void functions calls return a type return value by using a return statement. During the abstraction, the execution of the function should not end just as one reached the last line of it. Instead, CBMC introduces a assert (false) at the end of the function that ensures that there is not an error trace when location is reached. BREAK and CONTINUE statements are also transformed into GOTO statements as it is also SWITCH blocks.
The usage of loops is basically why a model checker need to be bounded. Here the transition of the program is unwound to certain point. The depth of this unwinding is crucial. If a loop is not unwound enough there is a change that a bug could be hidden just in the inner iteration of the loop. In ANSI-C loops are presented when using FOR and WHILE statements. CBMC unwinds this loops automatically but not always is able to determine where the amount of unwinding is sufficient. It is possible to manually set a amount of unwinding to be done (--unwind n , n number of times to unwind). In many cases, CBMC can not recognise the limits an goes unwinding for a infinite time expecting the resources are enough.

* Assertions and Arrays

This bounded model checker also recognizes the assertion defined in standard ANSI-C. The ASSERT statement is transformed and takes a boolean condition. CBMC will check that this condition is true for all runs and during creation of counterexamples.

* Arrays and Unions

The arrays structures, including multi-dimensional and dynamically-sized arrays are supported by CBMC. For dynamic arrays this tools transforms it into a boolean equation that does not depend on a value of a constant but on the number of read and write times that the array was accessed. Therefore, CBMC checks the validity of the upper and lower bounds for the arrays including the ones with dynamic sizes. This checks if the sizes are not negative. If that is the case, CBMC would create a counterexample in which the size of the array is a negative value, presenting a violation of a property.
When using unions, CBMC provides support allowing the use of same storage for different data types. This tool share literals employed for representation of variables within union members. What is checked by CBMC here is the non use of unions for type conversion.

* Pointers and Pointer to functions

The use in pointers is quite common in programs. These are used for call by reference and for management of dynamic data structures. CBMC fully supports the usage of pointers under the standards of ANSI-C. This includes pointer arithmetic and pointer type cast. ANSI-C does not make sure that the conversion of a pointer to an integer and later to produce a valid pointer. However, CBMC does it since a misconstruction would hide an error while abstracting the boolean formula. When comparing pointers with the use of operators { >, <, <=, >= }, CBMC ensures these not to be used by inserting assertions.
With regards to Pointer data types as described in standard ANSI-C, CBMC makes sure that the object being accessed correlates the type of the dereferencing expression.
When using pointers to functions, it is clear that the function pointed to vary depending on the chosen inputs that are non-deterministic. CBMC asserts always that the right function was called.

* String constants and Dynamic memory

The implementation of strings is handled as an array of characters in the ANSI-C standards. Consequently, the use of strings then is observed as pointers to elements in a array. The same array bounds are considered by the model checker. An additional check done by CBMC is that it ensure that no element in the string is modified (i.e. written) by the pointer used to access string elements (characters).This is also done by asserting this action never done by the program.
Programs using dynamically memory allocation for arrays and structures are supporter by CBMC. This tools check for array bounds when MALLOC statement is used. This is done by ensuring that a pointer pointing to a object that is dynamic is doing so to an active object. It makes sure we are not using a pointer to a location already freed or that does not represents a static location. CBMC also check for the times of free memory taking place, then the program can not incur in freeing a memory location more than once. Afterwards, the bounded model checker will make sure all memory has been deallocated by the end of the program otherwise errors will be reported concerning the memory management leading into leakage.

2.2.4.2.- CBMC at work

In this section will exposed by a simple example and diagrams how the Bounded Model Checker works. The diagram on the left comprises the work flow of the bounded model checker. Here CBMC First parses the source code just as a compiler would do it. Then it is transforms into a control flow graph (CFG) which is representation of a boolean formula. This interpretation is then taken is then taken to the SAT solver and express as a conjunctive normal form (CNF).

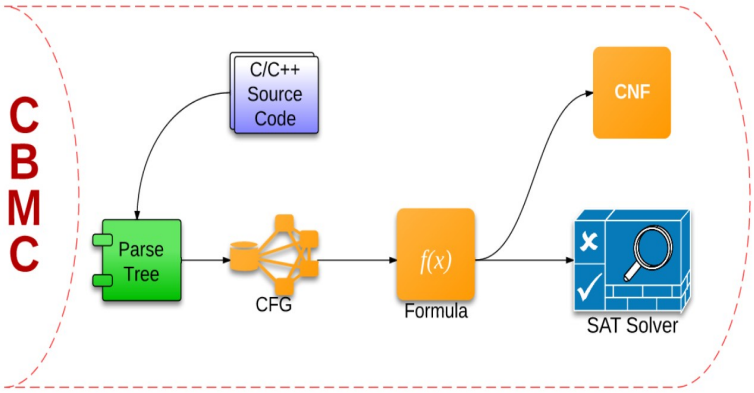


Fig.10. CBMC, flow of the bounded model checker.

On the Figure below, we can see how this powerful bounded model checker is applied to a short code and its outcome is shown on the right hand side of the screen shot. Note that CBMC does not need a starting point to undertake its bug-hunting procedure to start. In this small example, neither the values a nor b have been set with a numerical value.

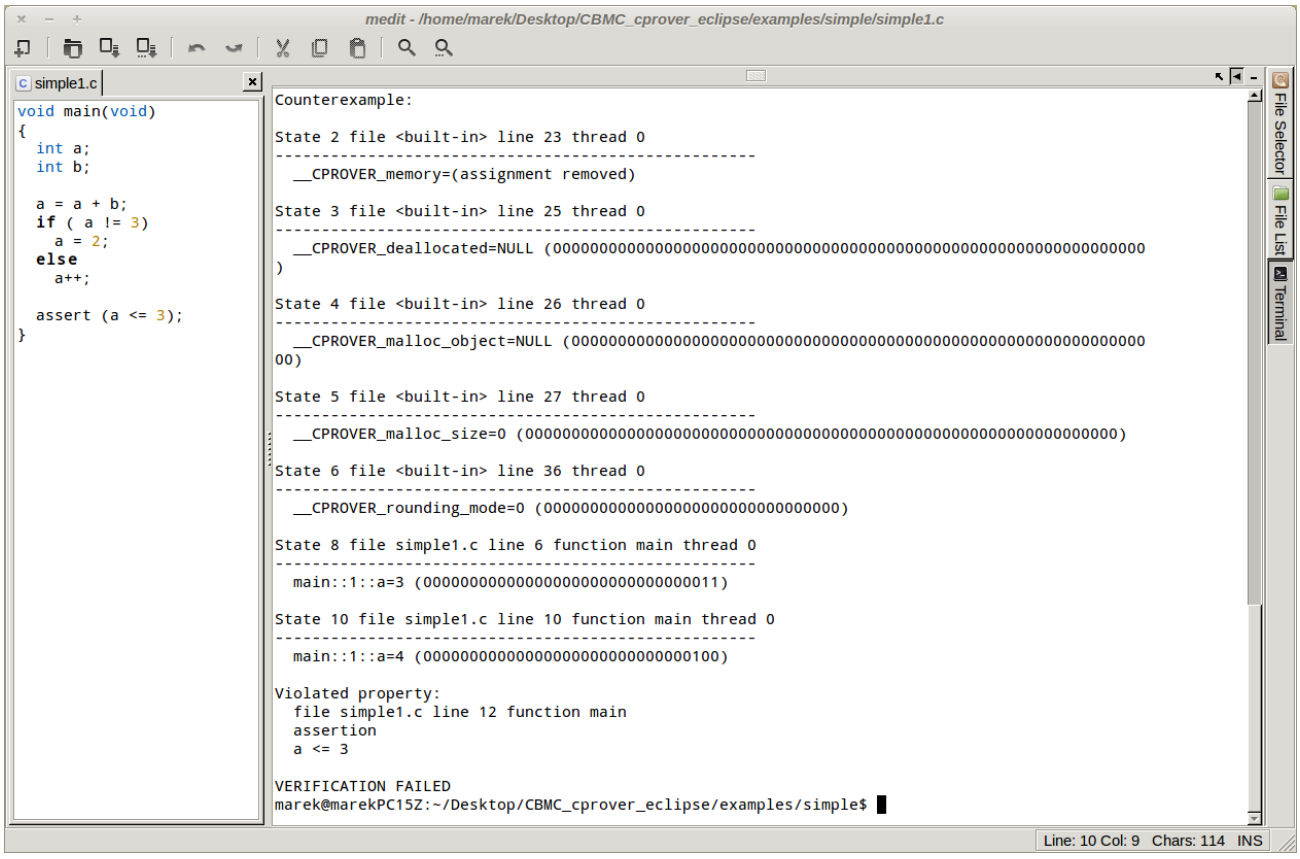


Fig.11. CBMC, flow of the bounded model checker.

2.2.5.- Summary of Bounded Model Checker - CBMC

CBMC has proved to be a powerful tool that enhanced the chances of discovering bugs that are the result of bad design, wrong format in coding and logic undecidability hidden in big project as in small code blocks as shown before. Hence, the usage of this tool will be of great advantage once the MPC is set ready for its verification and its translation process to a lower-level language description as C.

3 – MPC in high-level design for V&V

3.1.- Model predictive control in Simulink.

Model Predictive Control has been widely used in academia, and in the last decades, its use has seen an increase in the industry. As a result, various commercial software have been developed to cope with the demand of this control strategy according to the task. Between them, Simulink has an special toolbox that treats particularly the implementation of predictive control. Various examples have been set up using this tool. The design of the autonomous system is also implemented in Simulink. The code generator for the vehicle is set up in Simulink and the outcome of this is code in C language that handles the controllers of the autonomous system. The MPC modelling was provided by Dr. Arthur Richards from the Department of Aerospace Engineering, University of Bristol. The initial controller design is presented in the following subsection.

3.2.- Block description

As a point to start, a MPC model was used to examine the code generation of Simulink control system. This example was used in order to analyse the behaviour of the predictive control system and the implication of its constraints and the efficiency of the code generated. The main reason for firstly going through a basic example is that tools providing code generation are focused on performance and optimization[26]. Full completion on correctness of requirements of the code generated often is not reached. Here, it was considered a double integrator system set up in Blocks and hold in simulation for the generation of control code.

The initial script is "*mpcBuildSetpoint.m*", which was written in Matlab sets up the information data needed before starting simulation and test analysis of this example. After running this script, all data arrays and variable definitions will be ready, some basic constraints and error checkers are implemented for online report. The script is in appendix section A.

After having loaded these data, the Simulink model is ready to be launched. This provided simulation results (change of variables along the test) and also produced C code aimed for a embedded controlled system.

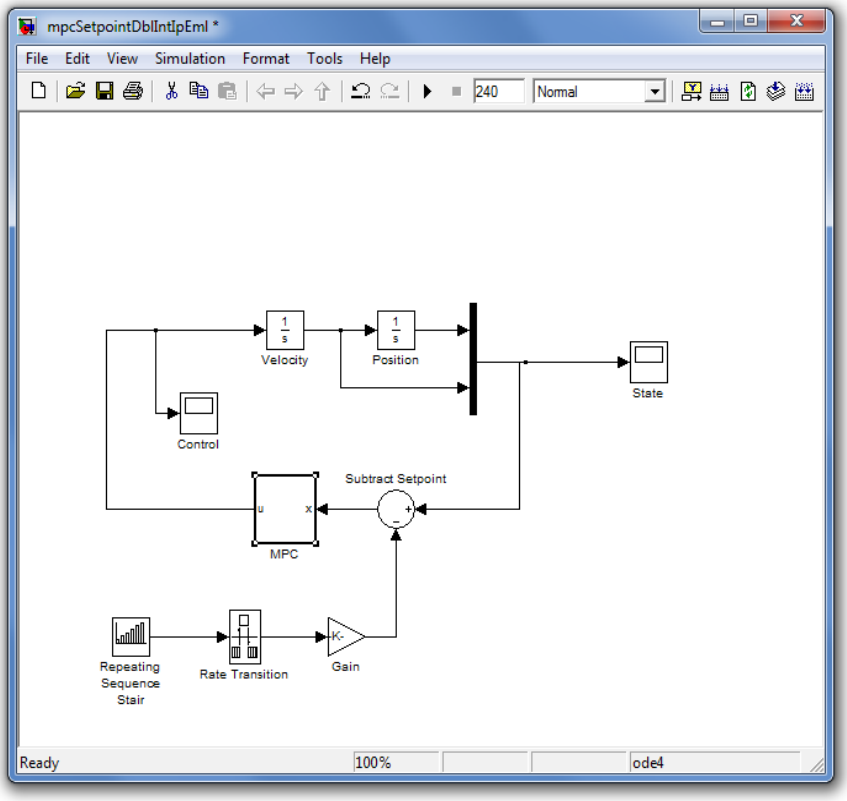


Fig.12. Structure of the MPC implementing C code generation.

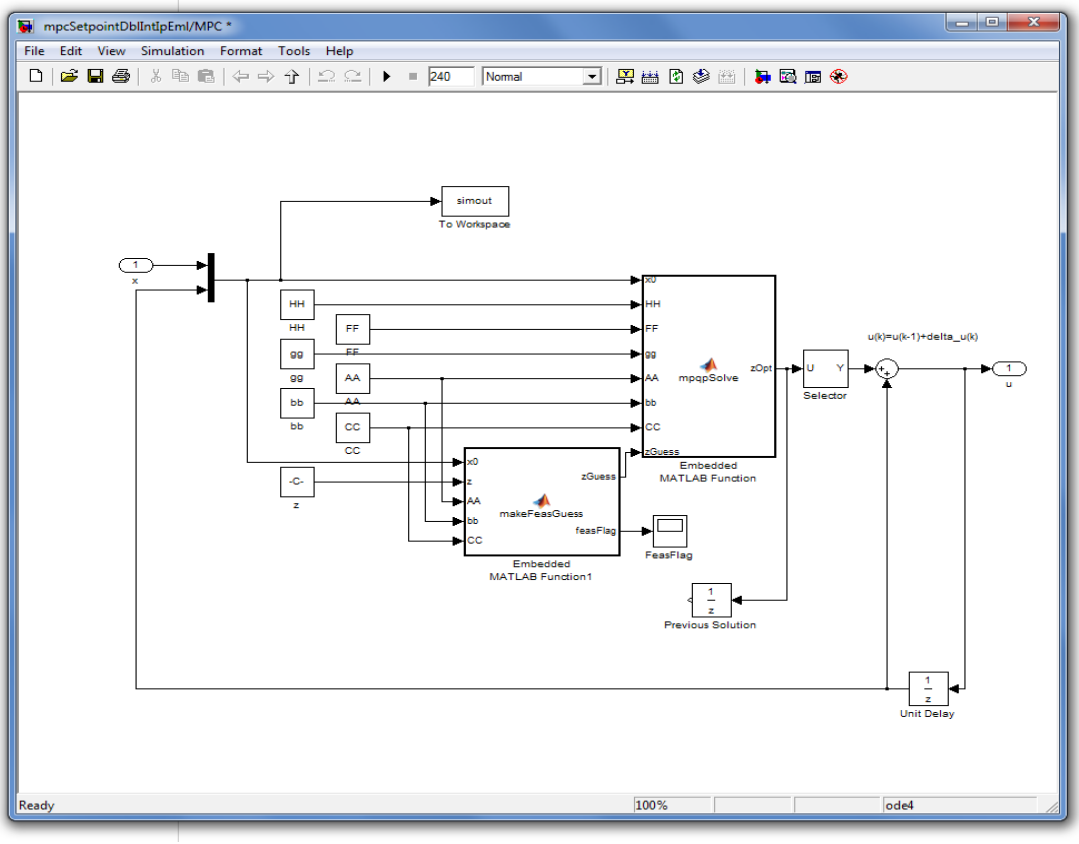


Fig.13. MPC core unit extended from fig 12.

The diagram above in solves the MPC problem minimizing a cost function. For this example in particular, "cost" is a articulation of quadratic combination of State and Control Moves.
The resulting quadratic problem was solved using a conjugate gradient solver plus a logarithmic barrier function. All these was implemented in Matlab blocks.

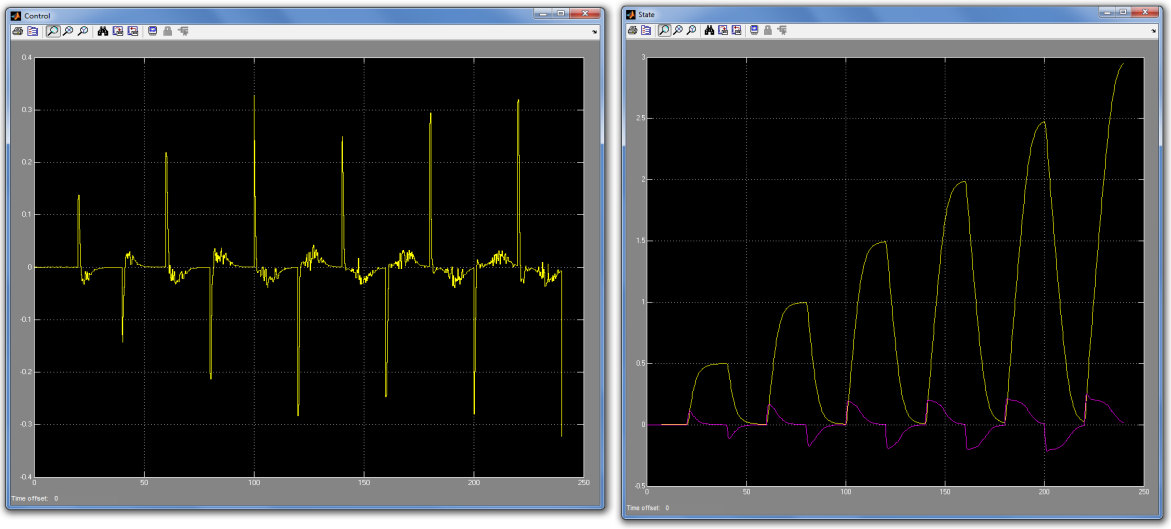


Fig.14. Above, the output monitor of Control and State.

During simulation, it was appreciated that the setting point gets bigger and reacts in a slower fashion as the velocity constraints become active in time.
The testing of the basic model provided us understanding of the functionality presented in the code generation. At the same time, it helped us to stablish a basic platform test for verification of a MPC system that is similar to the model employed in the autonomous vehicle. The emphasis here was the implementation of checkers in the way/method the MPC algorithm produces the low level code for the control of the robotic system. Further description is given in section 5 [Code extraction].

3.3.- MPC in the autonomous vehicle

The system under investigation is a robot vehicle in which MPC is employed for a controlled navigation system. This vehicles are used for test and development of control algorithms.
As it can see in the following picture, the vehicle has a computer on board as well as wireless network, an various adaptive motion sensors. If needed for the V&V process, further upgrades could be made, such as inclusion of GPS, cameras and extra sensors.

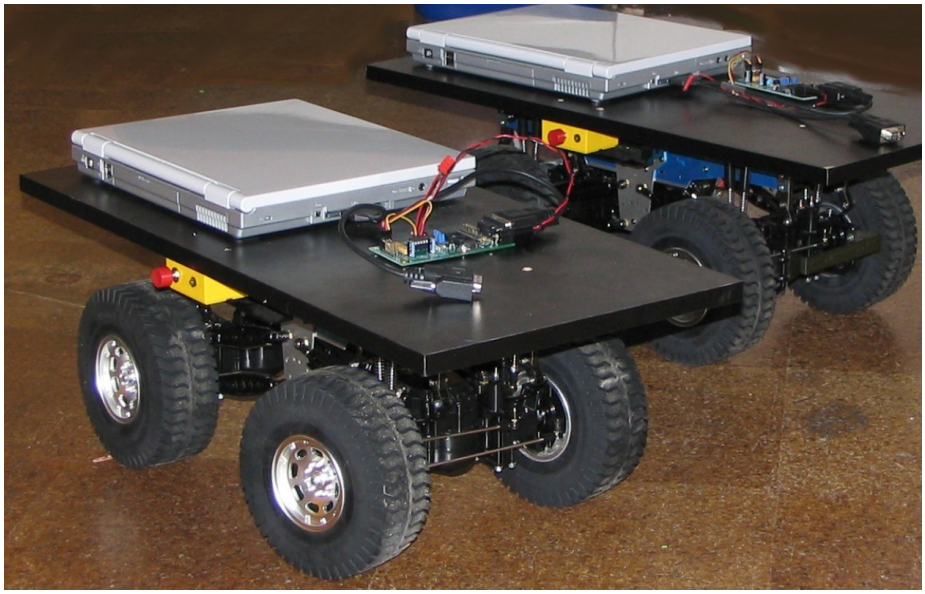


Fig.15. Test vehicles for autonomous navigation system.

The robotic system under test shown on the left, will be controlled with a predictive strategy. It will employ receding-horizon maximization of the miss distance, generating way points for a lower-level controller to follow.
In the strategy to be put in practice, loitering is implemented. Often called smart Loitering, this approach employs the flexibility obtained from manipulation of design's variables to modify the loiter path for a better separation. It will generate ordered way points which are then taken back to following controller. The algorithm is centred on the repeated solution of the following optimization:

$$\min_{v_1, k_1, \ldots, v_N, k_N} \sum_{i=1}^{N_i} -\log\left(\epsilon + \min_{\tau \in [0, N\Delta t]} \left\| r_{self}^{p}(\tau) - r_i^{p}(\tau) \right\|\right) + \alpha \Delta R + \beta \sum_{k=1}^{N} (v_k - v_{nom})^2 + (k_k - k_{nom})^2$$

Where the term $\left\| r_{self}^{p}(\tau) - r_i^{p}(\tau) \right\|$ represents the closest predicted approach to intruder *i* over a receding horizon, covering *N* times *Δt* in the future.

The optimized $r_{self}^{p}(\tau)$ is taken back to the way point controller. This method is widely used in the control systems for obstacle avoidance.

The control algorithm used in the autonomous navigation system works as the Smart Loitering. This system is flexible and integrates quite well in terms of constraints handling.

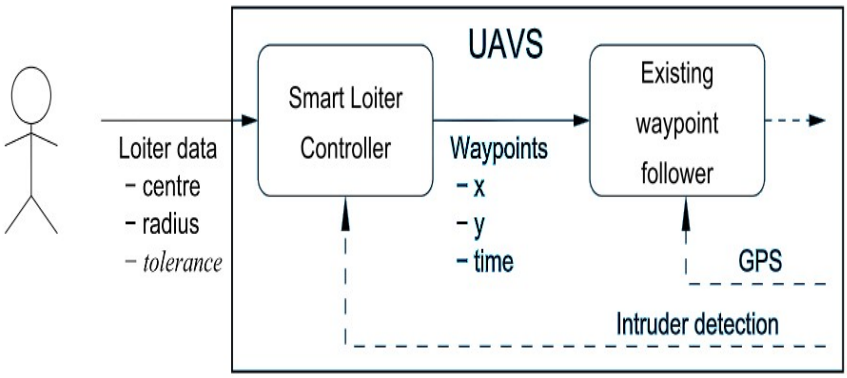


Fig.16. Smart Loitering system integration[6].

The above diagram shows in simplified manner how this predictive control strategy can be applied and integrated into UAV (Unmanned Aerial Vehicle). Due to that fact that Smart Loitering is a "reliable" predictive control compared to the a basic MPC, the way of handling constraints is crucial. Moreover, implementing the predictive algorithm and utilizing the code generator tool will expose more the design complexity and enhance the chances to perform a verification in a per module basis.

3.4.- V&V of MPC in the autonomous vehicle

The exploration and implementation of V&V process of this design considered the points mentioned before and also include ways in which the code generation could be verify. Depending on the complexity and reliability of this scheme for producing code that could be tested, the verification process attempted to formalize verification properties at different levels in order to compare results and see where it is possible to improve the specification or rearrange the constraints.

Despite the fact that the DUV (design under verification) is a autonomous vehicle (hardware), it is clear that its features and properties lay on the software ground. During the deployment of test benches, there was special software considerations in order to satisfy the Airborne Systems and Equipment Certification (DO-178B).

Consequently, emphasis in the software development process is crucial for any V&V. In this case, given that the design was provided by the avionics department, it was important to understand how strict are the regulations with regards to safety-critical software, which cover all the software development. In other words, apart from investigating the predictive control strategy employed, special concern regarding the software development process was studied. The latter is crucial for a proper selection of verification technique for autonomous system.

In the following section, it will be explained the method which may be used in the verification of the code generator, the predictive control itself and it will be presented the tool for analysis of test-bench for the design in high level block-model.

	Core Risks	Area	Description
1	Faulty Software Schedule or Budget	Planning	An error in the schedule or budget rather than in the software development and testing process; usually an underestimate of the cost and schedule.
2	Software Requirements Inflation	Requirement & Design	Significant increase in the number of software requirements over time.
3	Software Employee Turnover	Staffing	High number of programmers leaving a project – often not considered in budget estimates.
4	Software Specification Breakdown	Planning	A breakdown in negotiation; the majority of stakeholders can't agree on what they are building.
5	Poor Productivity	Experience & Teaming	Under performance due to an insufficient number of programmers with the required skills and experience.

Fig.17. Project risk due to software development issues[10].

3.4.1.- V&V in robotic systems and Software considerations

It is often said that the success of the project - meaning efficiency satisfied and safety covered - as a systems is predicated upon the ability to correctly verify and validate the software technologies, the hardware elements and the integrated system as a whole. It is imperative here to differentiate these both terms.

Verification is the process used to determine whether or not a system satisfies a given input and outputs, e.g. whether the implementation is correct according to the specifications. Validation is then used to determine whether these input and outputs function actually has the desired behaviour when the system is deployed (in practice). Interesting aspects of validation of complex systems are exploration and confirmation of emergent system behaviour in the target environment and associated risk assessment. In particular, using models and simulation in this context requires determining how accurate these reflect the real world, how much the models can be "trusted" which permits conclusions on the derivations made from them, whether the models are good enough for us to detect safety violations, etc.

This project focuses on to the code generator employed by the predictive control. As it can be seen in figure 17, the table shows the different risks that may arise due to the software development. Knowing these factors, better possibilities will be conceived to use advanced software technologies and state-of-the-art embedded software architecture. Additionally, in the process of verification and validation a modular analysis was performed. These steps were critical for study of the system as a whole by undertaking a verification by block.

As presented in [10], the software life cycle is important since it may lead to the exposure of flaws in the design, unclear constraints or unnecessary conditionals.

LIFECYCLE PHASE	V&V TECHNIQUE
System Requirements	Requirements Consistency Analysis
System Design	Requirements Consistency Analysis
Software Requirements	Requirements Modeling/Analysis
Software Architectural Design	Architectural Design Modeling/Analysis
Software Design	Detailed Design Verification
Software Coding	Auto-Code Generation
Software Unit Testing	Auto-Test Case Generation

Requirements Consistency Analysis : requirements definitions
Requirements Modelling/Analysis : static or dynamic analysis at the requirements level)
Architectural Design Modelling/Analysis: architectural design for consistency
Detailed Design Verification:
Verification of safety and "liveness" properties
Verification of parameters
Validation of the design model during simulated execution
Detection deadlocks and timing violations
Detection of concurrency errors

Fig.18. V&V techniques in software life cycle phase [10].

Auto-Test Case Generation: generates test scenarios based on limit values. Here automatic checkers oversee the correctness and report violations of minimum and maximum values accepted in the system.

Auto-Code Generation: At this stage (software coding) for the autonomous system, the implementation of checkers for error detection takes place during the code generation and also it enhances the chances to prevent defects in the initial design (high level Simulink block model design).

The tools to be employed work such that if the design is faultless, and on top of that the verification of the design is automatic, then the code produced by this tool will "defect-free". Despite the drawbacks brought by this tool, the advantages are notorious. One of them is ability to trace requirement (design specifications). It is then possible to track the fulfilment of a particular rule, constraint or generic specification. Another advantage is the chance to check for completeness of design. This is quite important if it is intended to stablish some coverage target. This will lead to better approach towards the checking of the model. Also the auto-code generation does provide the chance to undertake debugging of the system at a design level. Using all these properties will improve the verification at this stage.

As mentioned previously, safety-critical systems cannot afford much relaxation of constraints. In order to obtain a certification or safety approval (e.g. by industry body) rigorous and strict measurements are required. In order to prove that the expected result eventually happens, it is often needed to prove that a wrong/unexpected result will never happen. Meaning that to prove a liveness property it is needed to proved more than one safety property. A number of methods are proposed for verification of safety properties[32,33]. Is here that the use of assertions will show it capabilities [28]. Assertion-base verification (ABV) can be extremely effective but quite tricky to implement.
The so called partial correctness property employs assertions and from a safety point of view, using this partial correctness exemplifies the simplicity of an assertion but it also involves multiple undefined states within the system, making it irregular in many cases [33].
Assertion example:

> if a program program S begins execution with some precondition P true, and if the execution of S terminates, then it terminates with some postcondition Q true.

$$(at\ S \wedge P) \supset (after\ S \supset Q)$$

For this assertion to be satisfy at all times, the logic presented above will deal with corner cases which could not be foreseen at model design stage or during test-bench implementation.
Given that the DUV employs a predictive control strategy that is sequential (not fully concurrent), during the study of the model, the intermittent assertion method [32,34] was used for the different test-benches needed in the V&V process. Ultimately, a global test-bench more alike of a hybrid control system will be needed to demonstrate the validity of the system.

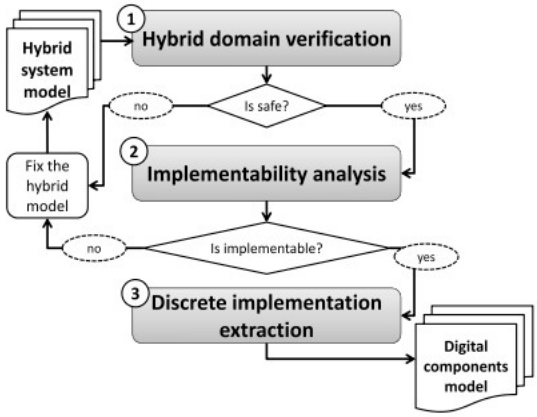


Fig.19. Test bench method [26].

3.4.2.- V&V for complex systems

In terms of system integration, the verification and validation was given for an embedded software (ESW) project. At this stage, the global automation of the test-bench is aimed. A final prototype is presented that shapes the nature of the predictive system control as part of a global verification system that employs the code generated to satisfy the specifications.
Here, the verification cycle could have been commanded through a PSL (Property Specification Language) [37], because of its flexibility. Furthermore, corner case oriented mechanisms could be implemented during the simulation of the ESW and checkers will be generated to achieve high coverage test scenarios.
This verification system has been built as a model-driven design, which raises the level of abstraction. This method is appropriate for complex systems[35], and it does implement support for code generator tools (code generated in C). The Unified Modelling Language (UML) [36] has the advantage of a more visual modelling. Its use has been quite popular. Nevertheless, it does not guarantee specification correctness.

The predictive control strategy employed in the robotic system is not just complex to be formalise but also not static. As shown in the previous section of code generation using Simulink (section 4.1), a predictive control algorithm can only be "observed" through its constraints and manually implemented checkers. Consequently, it would be impractical to set up checkers at a high level of the design.
After the definition of the PSL properties, the formal properties could be automatically synthesized towards a correct by construction executable checkers. This will be achieved if a fully integrated and dynamic checker engine is implemented in the verification system.

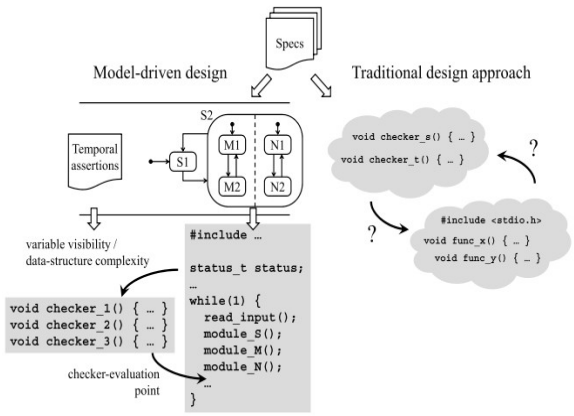


Fig.20. Model Driven design Design and Validation flow [28].

3.4.3.- V&V tools used in MPC

The tools to be used for the verification and verification of the test-bench to be implemented is the V&V toolbox. The final DUV (device under verification) is presented in section 4. It incorporates a predictive control with a similar smart loitering approach, widely used in unmanned aerial vehicles(UAV).
However, tools and operability were investigated in order to select the most appropriate ones for the task.
In Simulink - Matlab- the tool box "Simulink Verification and Validation " does provide a great functionality with respect to the safety-related test. As mentioned in the product documentation, this tools offer a special functionality for Software considerations in Airborne Systems and Equipment Certification referred as DO-178B Checks.
"*DO-178B Checks Overview* DO-178B checks facilitate designing and troubleshooting models from which code is generated for applications that must meet safety or mission-critical requirements. The Model Advisor performs a checkout of the Simulink Verification and Validation license when you run the DO-178B checks." [11]
This tool has been studied and tested in order to make feasible the implementation of a test-bench with all the special features from the toolbox mentioned above and the code generator toolbox as well. Simulink Verification and Validation set the requirements tracing to be automatic. Test-harness generation and modelling standard compliance checking are also automatically checked. This V&V tool supports standards checks for modelling in the DO-178B and IEC 61508 industry standards. The main checks are as follows:

"

-Check safety-related optimization settings
-Check safety-related diagnostic settings for solvers
-Check safety-related diagnostic settings for sample time
-Check safety-related diagnostic settings for signal data
-Check safety-related diagnostic settings for parameters
-Check safety-related diagnostic settings for data used for debugging
-Check safety-related diagnostic settings for data store memory
-Check safety-related diagnostic settings for type conversions
-Check safety-related diagnostic settings for signal connectivity
-Check safety-related diagnostic settings for bus connectivity
-Check safety-related diagnostic settings that apply to function-call connectivity
-Check safety-related diagnostic settings for compatibility
-Check safety-related diagnostic settings for model initialization
-Check safety-related diagnostic settings for model referencing
-Check safety-related model referencing settings
-Check safety-related code generation settings
-Check safety-related diagnostic settings for saving
-Check for model objects that do not link to requirements
-Check for proper usage of Math blocks
-Check state machine type of Stateflow charts
-Check Stateflow charts for ordering of states and transitions
-Check Stateflow debugging settings
-Check for proper usage of lookup table blocks
-Check Stateflow charts for uniquely defined data objects
-Check usage of Math Operations blocks
-Check usage of Signal Routing blocks
-Check usage of Logic and Bit Operations blocks
-Check usage of Ports and Subsystems blocks " [39]

4 – Implementation of Test-bench of MPC in Simulink

In this section, it will be explained the implementation of test-bench by inserting assertion according to the specifications of the MPC algorithm.

4.1.- Setting up MPC constraints

After studying the algorithm, the specifications of the systems have some rules to be hold at all times during simulation. These are then represented in constraints. This constraints handle the boundaries of the system while working at its limits.
The model is physically constrained by :

Velocity Control :	`abs(v(k)) < 0.2`	(soft constraint)
Force Control:	`abs(u(k)) < 0.5`	(soft constraint)
Movement Control:	`abs(u(k+1)-u(k)) < 0.5`	

Having in mind the previous constraints, there is another important feature of the MPC system. This is related to its optimizer. For the MPC to work as expected, we hope the optimizer makes the best prediction or at least not violate a property would prove that this block is not correctly predicting the best input even knowing some past inputs. This will be explained in the block diagram further in this section.

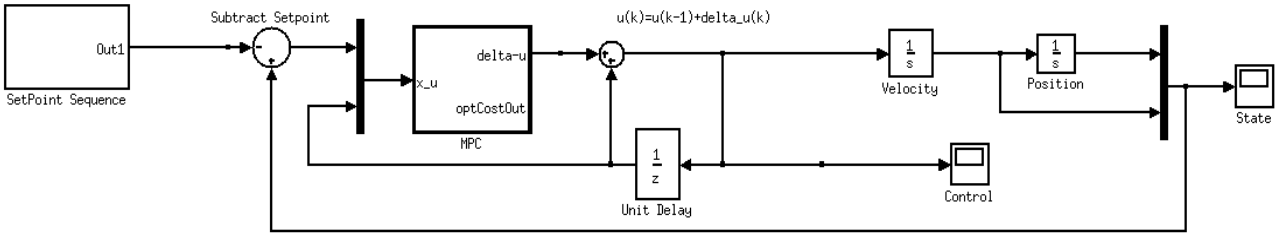


Fig.22. MPC to be verified

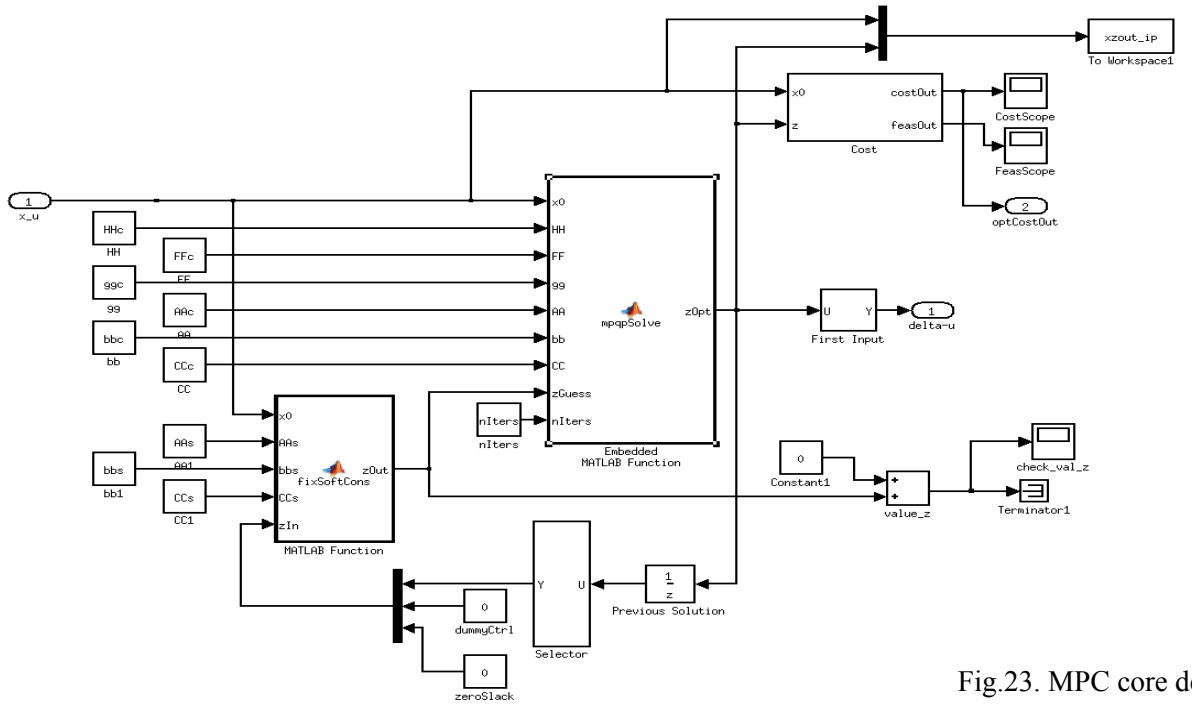


Fig.23. MPC core descroption

The blocks “fixSoftCons” and “mpqpSolve” are found in appendix section A.

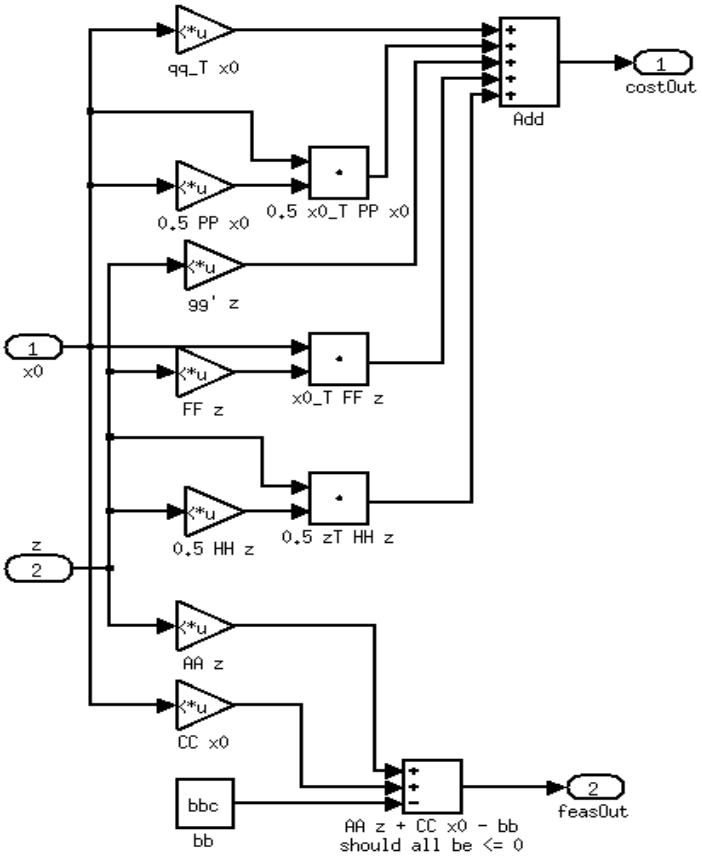


Fig.24. Cost Optimizer block model

On the left, it is shown the block description of the “Cost” block used in the MPC core.
As we are trying to implement checker for this system, a global perception is needed in order to see where it is possible to implement the assertion(s) to build a test-bench and get some results.
The constraints were inserted in a simple way by using simple logic blocks to keep the values of the movement, velocity and force under control limits. This is shown in the Figure 25.

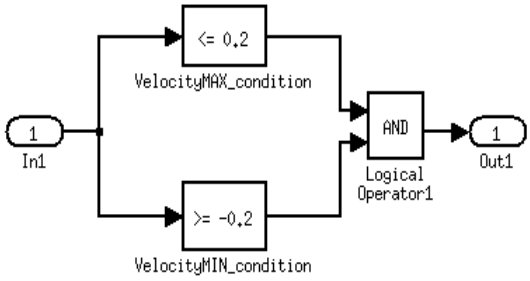


Fig.25. Cost Optimizer block model

The property blocks implemented in the block design makes sure that the conditions for the velocity, movement and forces are fulfilled. Just after that, it is possible to insert an assertion that will ensure that this construct is true at all time during runtime-verification. It is also added a “Signal to Workspace” that will carry the output of the property blocks that will be checked in the code generated. This is shown in the following Figure.

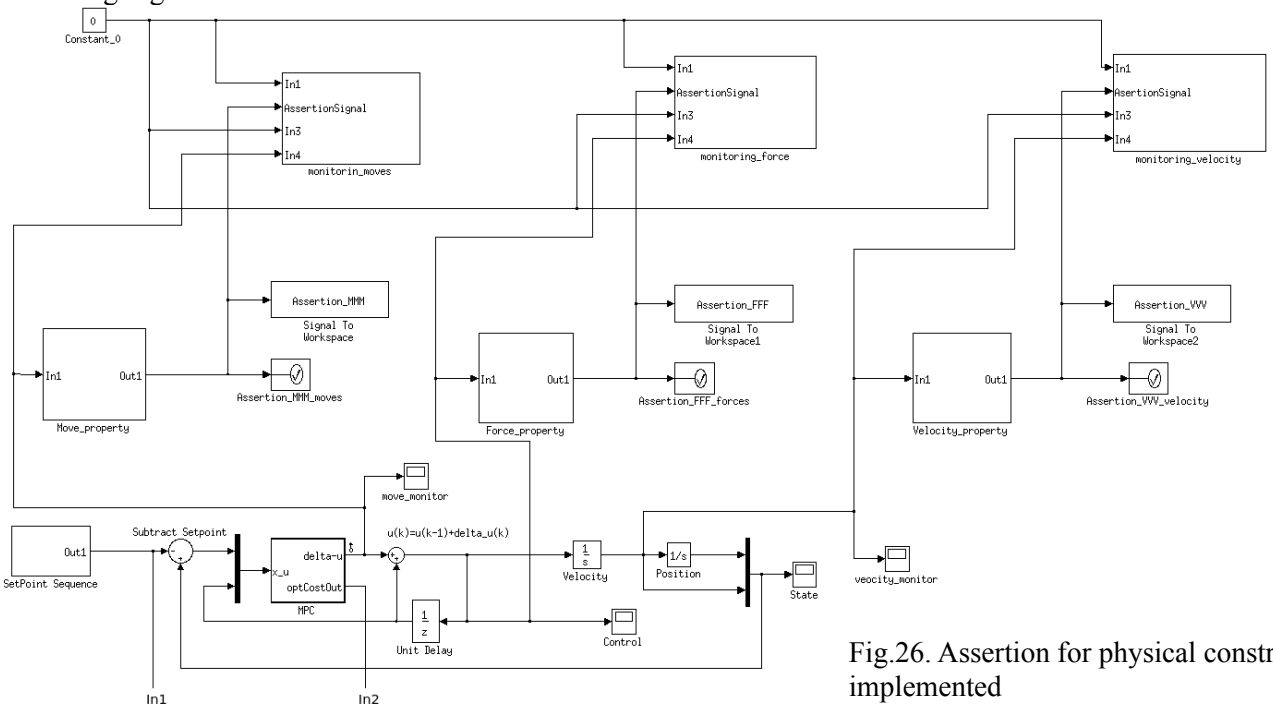


Fig.26. Assertion for physical constraints implemented

From the previous Figure, it is seen three block on the top. These are just monitor that were set up in order to corroborate the values of the assertions, states and variables during simulation. The Move_property, Force_property and Velocity_property blocks contains the conditional needed for the assertion. Assertion_MMM/FFF/VVV blocks triggers the call of assertion for their correspondent property block. Just on top of the assertion block there is a signal to workspace which will be added in the vector output of the MPC system. Below the MPC block we can see there are two unconnected paths In1 and In2. These are shown in the following figure.

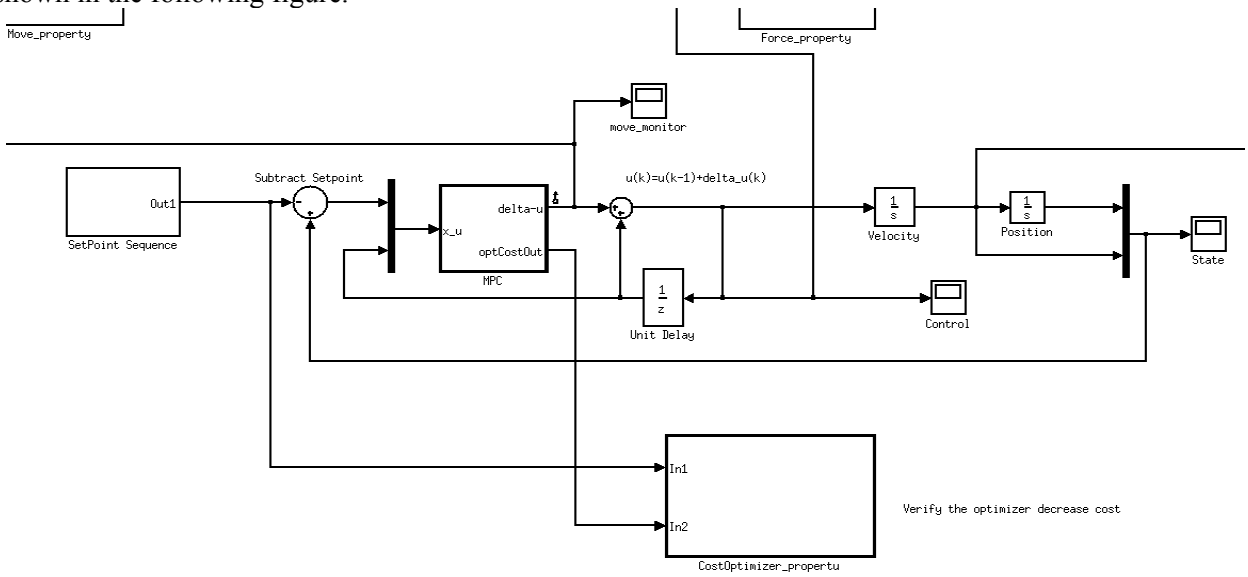


Fig.27. Assertion for physical constraints implemented

The "CostOptimizer_property" block will ensure correctness of the optimizer inside the MPC control. The Inputs taken by this block are state and cost of the MPC system during simulation. What is checked here is related to efficiency of the optimizer. If the one state is equal to the previous one(Sp_current == Sp_old), then make sure that the current cost is less or equal to the last one (J_new==J_old) .

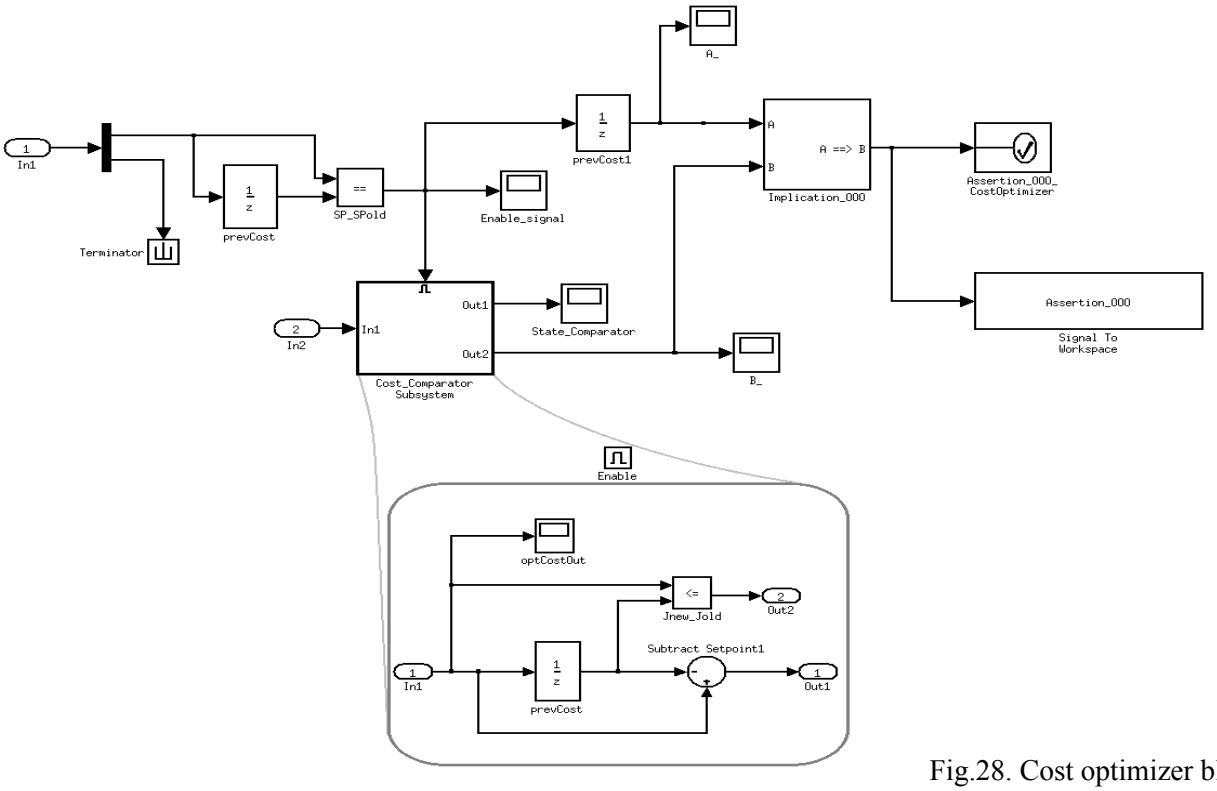


Fig.28. Cost optimizer block property and assertion of conditionals

In the Assertion_OOO_CostOptimizer it is checked whether the conditional for the cost function is hold at runtime-verification. In the same way as done in the other three property implementation, here it is also added a SignalToWorkspace to be used when tracing this assertions. Note that this property block defines its conditional by using an implication A ==>B. Moreover, this conditional will be triggered only when a secondary block is enabled for further comparison.

4.2.- Simulation outcome

Before running the first test we need to ensure to previously load all variable for the set up of MPC algorithm. This is done by running the script and then modified the condition to continue simulation if an assertion fails. The following Figure shows the waves of the signals A_ and B_. It is clear that there are some minor error in the cost optimizer.

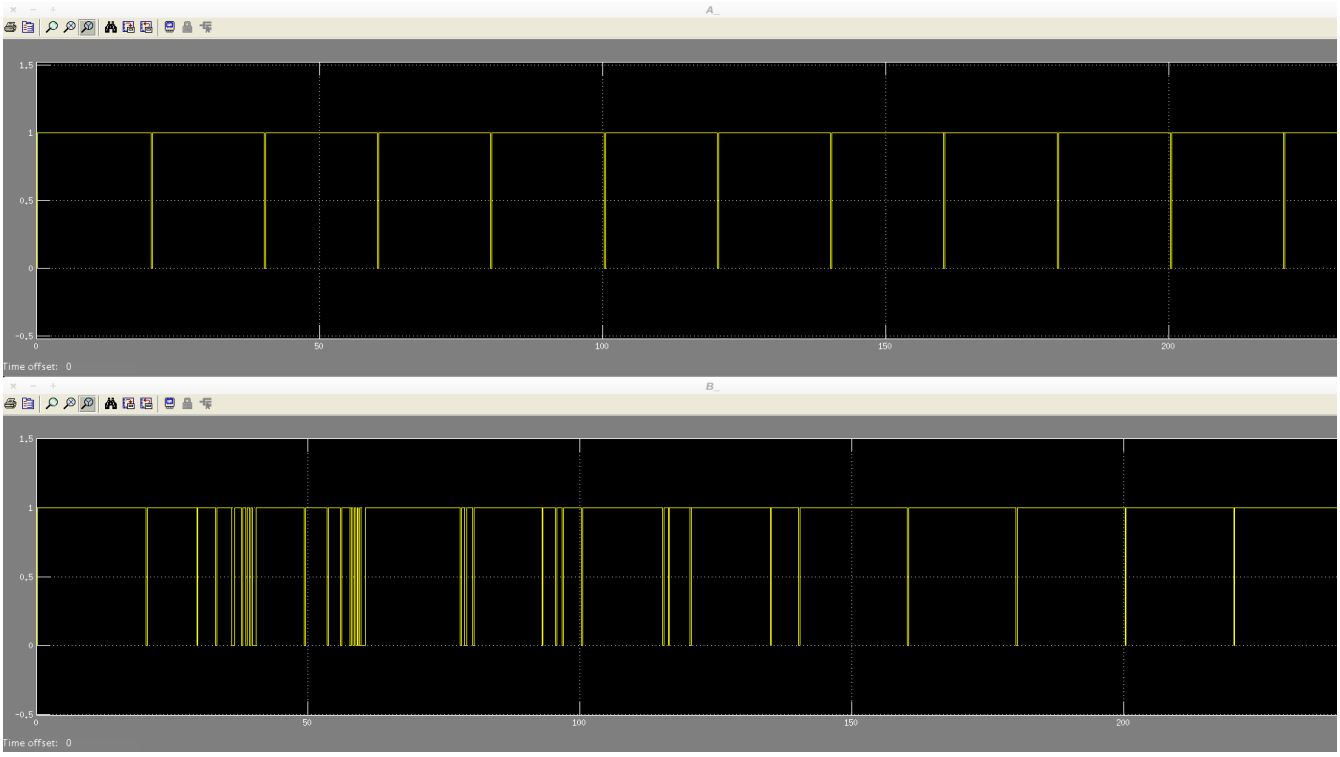


Fig.29. Signals A and B used in the cost optimizer property.

Despite the fact that the Optimizer fails during simulation, apparently the functionality of the model does not look as having a major control problem. This can be appreciated in the following Figure that describes

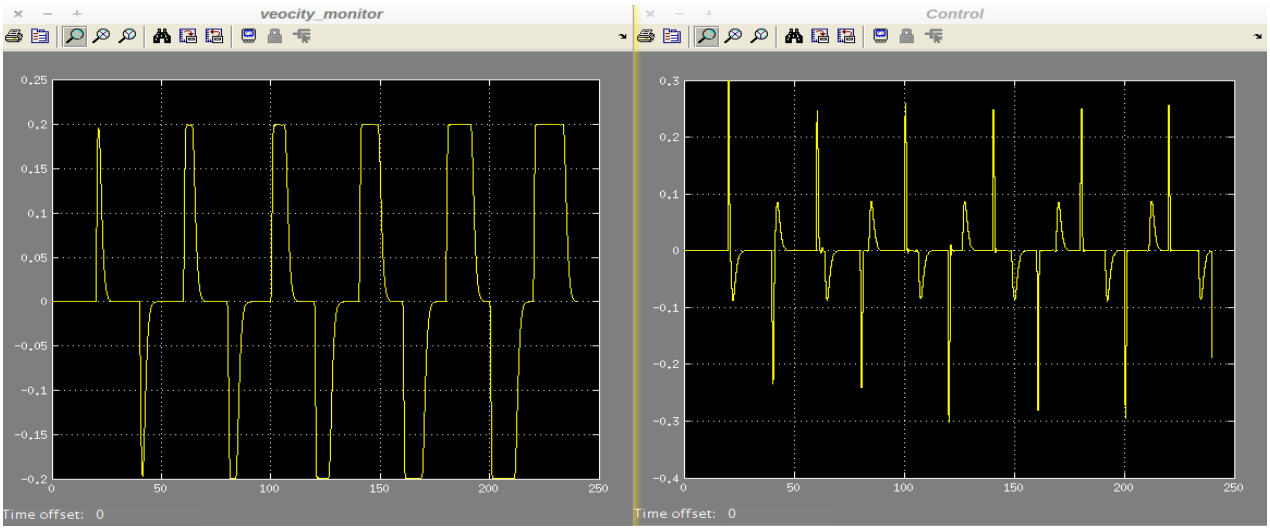


Fig.30.Left plot describes the wave of velocity control while the right plot is the movement control variation.

the waves of the control movement and control velocity. Both of these seem to work within the boundaries set with the property blocks and insertion of assertions.
However, during simulation in the Matlab's command prompt it was reported two time violation of the velocity property besides the cost optimizer property that reported larger number of violations represented as warnings. The complete output can be seen in Appendix section B.

4.3.- Setting up the coverage for System.

In the block model the coverage can be set to a particular blocks. In this case it was considered all the project. Meaning that all blocks down in hierarchy were considered while running the tests and used by the Simulink Verification and Validation toolbox. The following figures shows which block were covered. The uncoloured blocks are just "wires" static paths, therefore not considered by the toolbox.

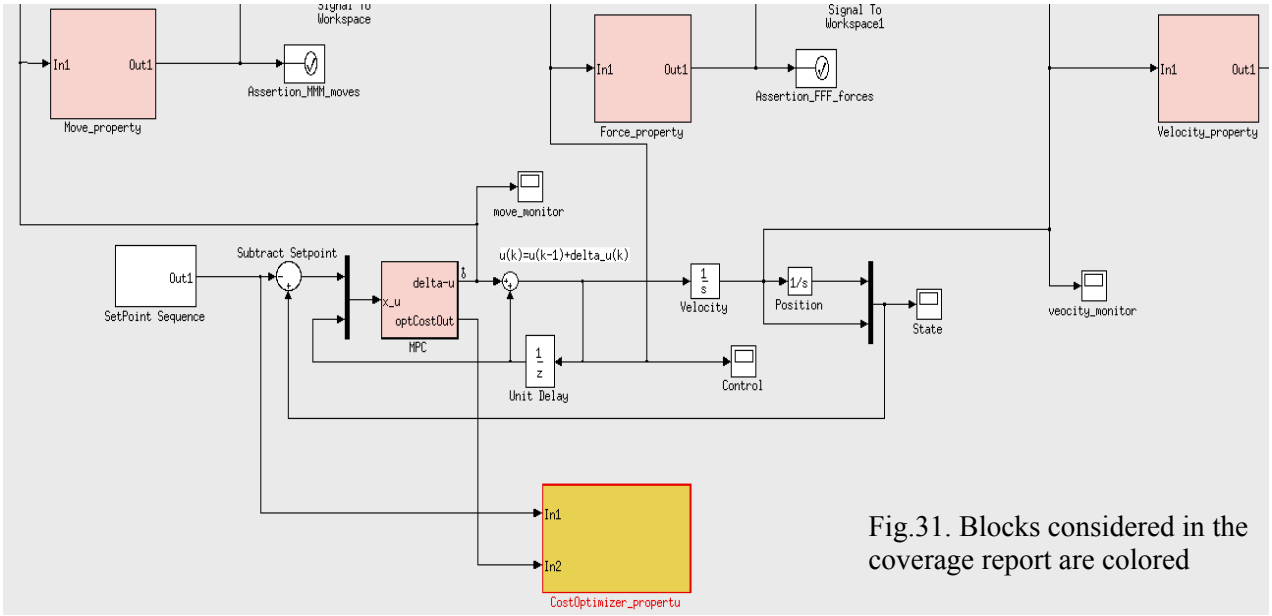
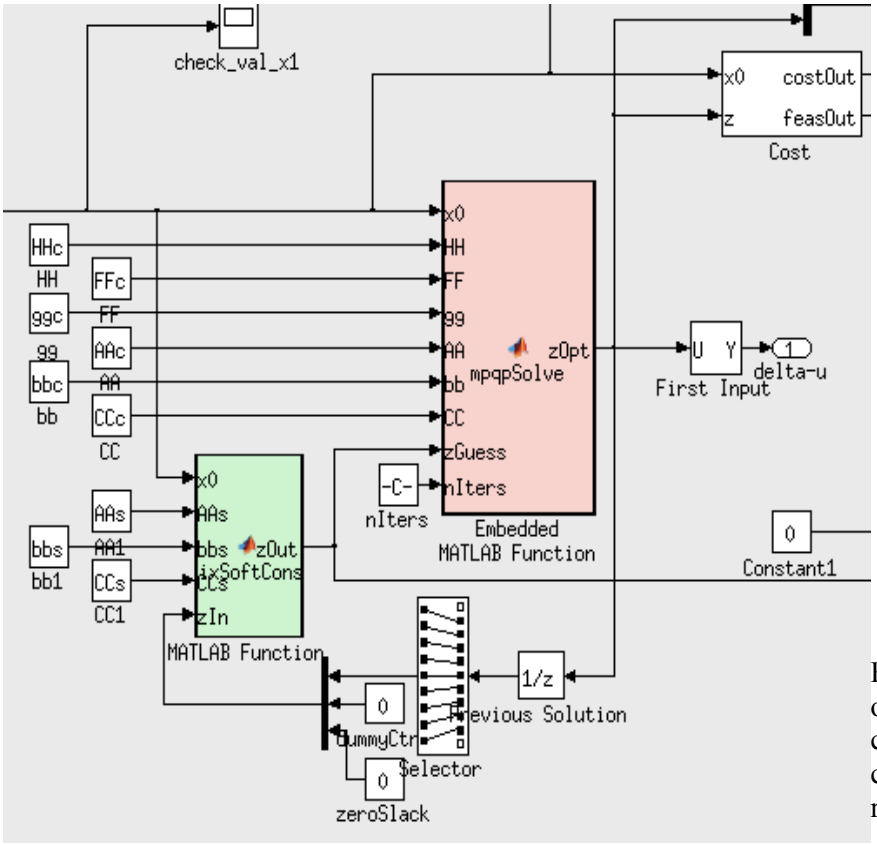


Fig.31. Blocks considered in the coverage report are colored



The conditionals inside the property blocks are definitively covered since they include paths of analysis for the value of movement, velocity and forces control. In the same way, the V&V toolbox recognizes that the CostOptimizer_property contains conditionals. The MPC block is also covered. Here both script are included in the coverage, "fixSoftCons" and "mpqpSolve"

Fig.32.In the core block of MPC, only these coloured blocks were considered in coverage report.

5 – MPC model translation - Code generation

This sections explains the procedures taken to have the source code in C language built. The abstraction was done in Microsoft Visual Studio 2010 Professional (MSVS) V4.0.30319. Afterwards, description of the files to be obtained in order to have the project in a single directory is explained.

5.1.- Building and Running the mpcSetPointIntpt project

The first thing before building and running the mpcSetPointIntpt.ml file from Simulink is to configure some options. These are located in Tools > Code generation > Options... in the window of the mpcSetPointIntpt block diagram project as shown below. Right after that, it will be needed to modify the option target selection. This can be seen in figure 34. In the section of Code Generation, there is a section where it is possible to specify what type of target is going to be considered while building the C project.

Here, we selected *grt.tlc* , but making sure that this Creates a Visual C/C++ Solution for the "grt" target.

Then it will be important to check again if all blocks are correctly named because it will be needed for traceability of blocks into code structures.

In the Code Generation, there are quite useful options that could be selected to make easier the tracing of the system in the low level language. (e.g. Matlab source code as comment, reuse and keep variables name, etc.).

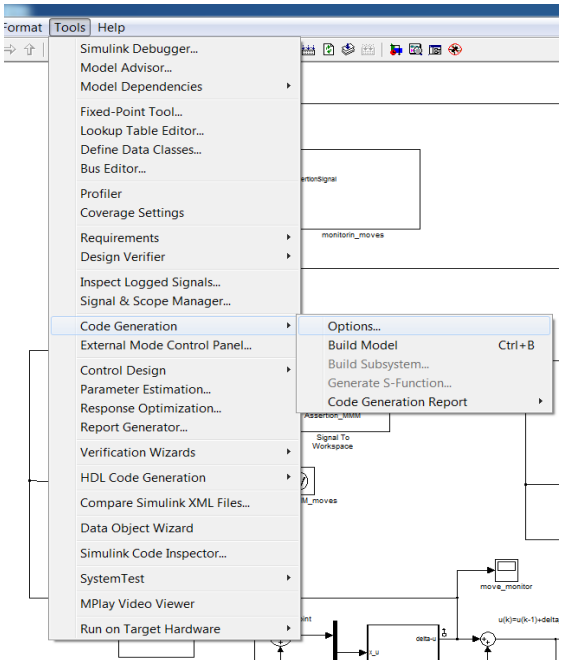


Fig.33. Selecting option features of code generation toolbox.

The selection of the proper grt.tlc is quite important. It has be selected from the browse button. The output of the building process in Matlab can be seen in Appendix section C. The outcome of building in MSVS can be seen Figure 35.

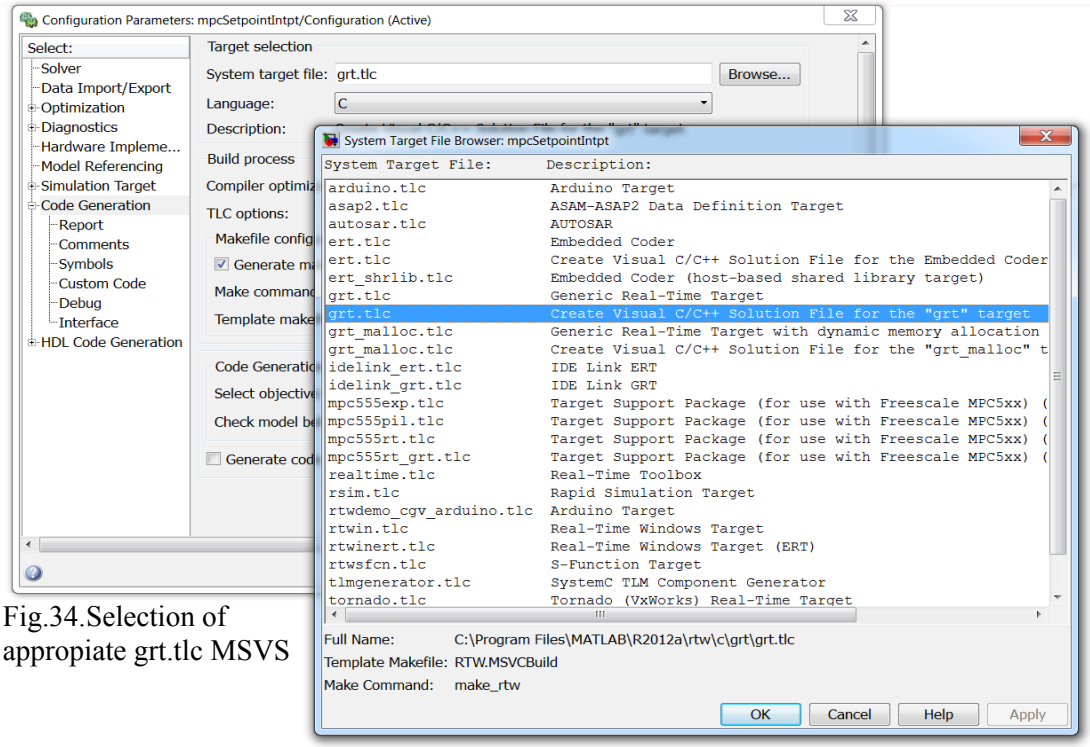


Fig.34.Selection of appropiate grt.tlc MSVS

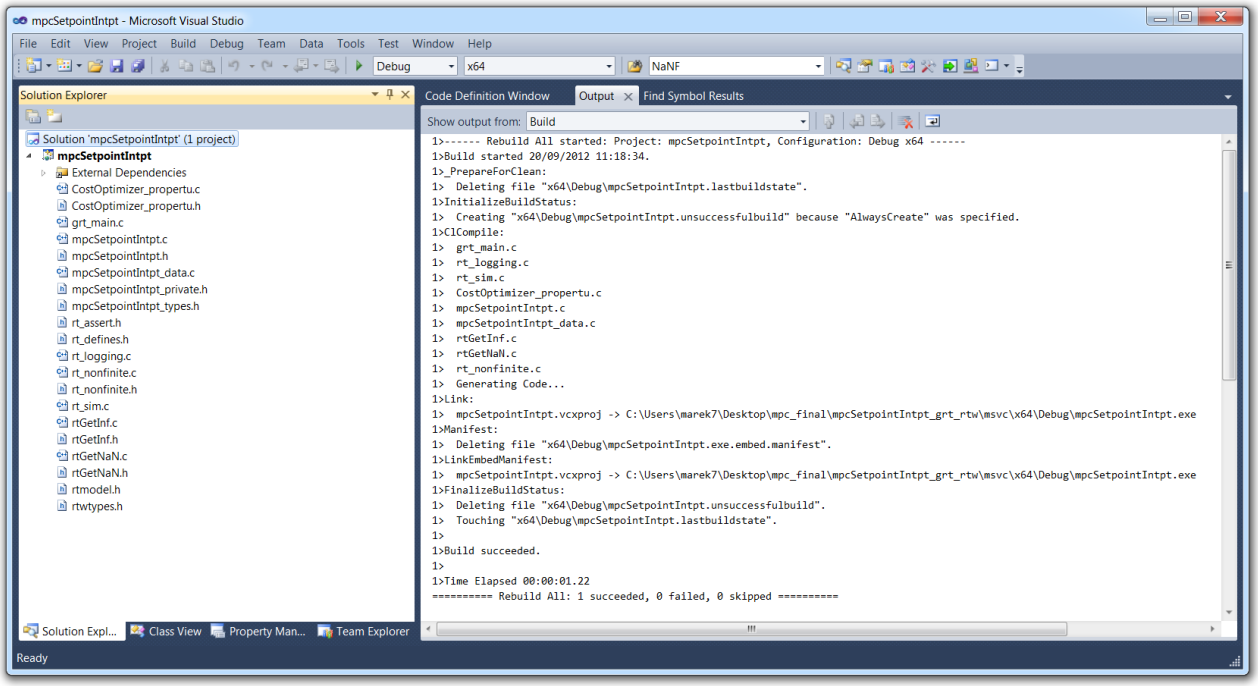


Fig.35. Outcome for building the MSVS solution of the MPC model.

The Figure above show the project built without errors in MS Visual Studio.
In order to have the code working, a project had to be created. The new project should use a WIN32 Console application from the section Visual C++ and set as a console application type. Then all files generated were added, these are *.c and *.h files from the directory ../mpcSetpointIntpt_grt_rtw that is located along with the .mdl file (file keeping all Simulink block model).
Afterwards, there are other files that must be include for correct compilation. These are in following subdirectories:

- C:\Program Files\MATLAB\R2011b\simulink\include\ *.c and *.h files
- C:\Program Files\MATLAB\R2011b\rtw\c\src*.c and *.h files
- C:\Program Files\MATLAB\R2011b\rtw\c\src\ext_mode\common*.c and *.h files
- C:\Program Files\MATLAB\R2011b\extern\include*.c and *.h files

For each directory, the content needed was copied into a new directories location and their properties were changes to allow modifications due to the fact that some of these files are read only. Another very important file needed is *grt_main.c*. As its name specifies, this file contains information of the complete project or solution called in MSVS. It is located in:

- C:\Program Files\MATLAB\R2011a\rtw\c\grt\grt_main.c

This code extraction can be done also in Linux platform, the location of files are the same except for the installation of Matlab (change the beginning of path: File System/usr/local/MATLAB/R2011b/....). An additional file needs to be added. This file is *define.h* that is the header which includes the preprocesor definitions. The header was included in the files *grt_main.c* and *simstruc.h* inside the project. This file can is described in Appendix section D. Preprocesor definitions were constructed by both the Matlab compiler and MSVS. Hence the necessity to add them for correct compilation of C source.

5.2.- Authentication of C source and mpcSetPointIntpt project

Once the files of the C project/solution are running, they were transferred into a Linux platform where CBMC works. Previous to this step. By deploying in an IDE Geany (v.021), the files were united in a project. The directory location is shown below. It is clear that all files were put into a single directory. The all_in1_sin location contains the files needed for the bounded model checker.

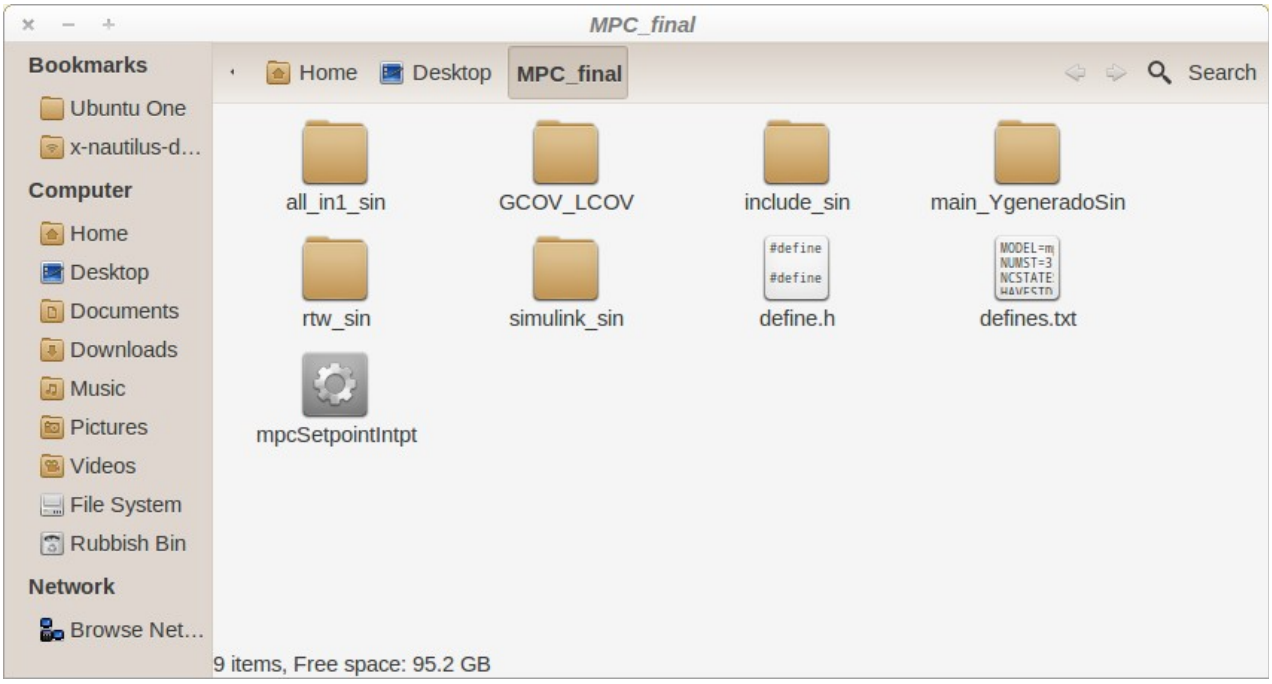


Fig.36.Directory arrangement for compilation of C source code in Linux for CBMC use.

This project then creates an executable that produced a *.mat file. This was compared to the *.mat file created from running the executable from Matlab. To run the executable in Matlab, the following command was used :

```
>> !./mpcSetpointIntpt
```

The response of this command was:

```
** starting the model **
** created mpcSetpointIntpt.mat **
```

The creation of mpcSetpointIntpt.mat took place and then compilation and execution of both ways were compared. A *.mat file is a Matlab format to keep information of vectors. These can be input and outputs, depending on what was set as a signal to the workspace in the Simulink modelling stage.
The output of both executables were compared to corroborate that the translation from the high-level model design does perform equally. This was done many time and started with simple block models non related to a control systems. It was checked the the content of these *.mat files were equal. The output of this file after execution can be seen in the following Figure. The created *.mat file contains nine different vector constructs. These all represent independently the output signals of the predictive controller.

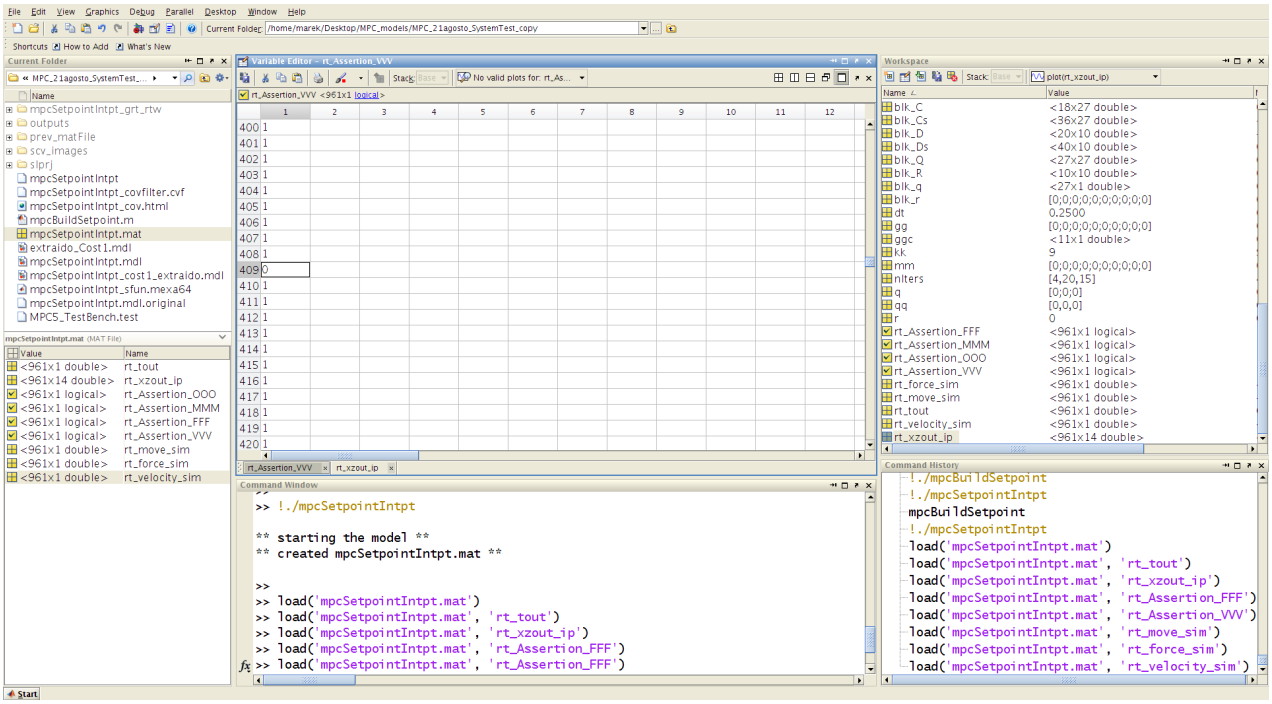


Fig.37.Screen shot of MATLAB presenting the produced mat file obtained after execution of model built in Matlab.

Below it is shown how after testing in a light IDE, the project was compiled in order to have it ready for a formal verification procedure.

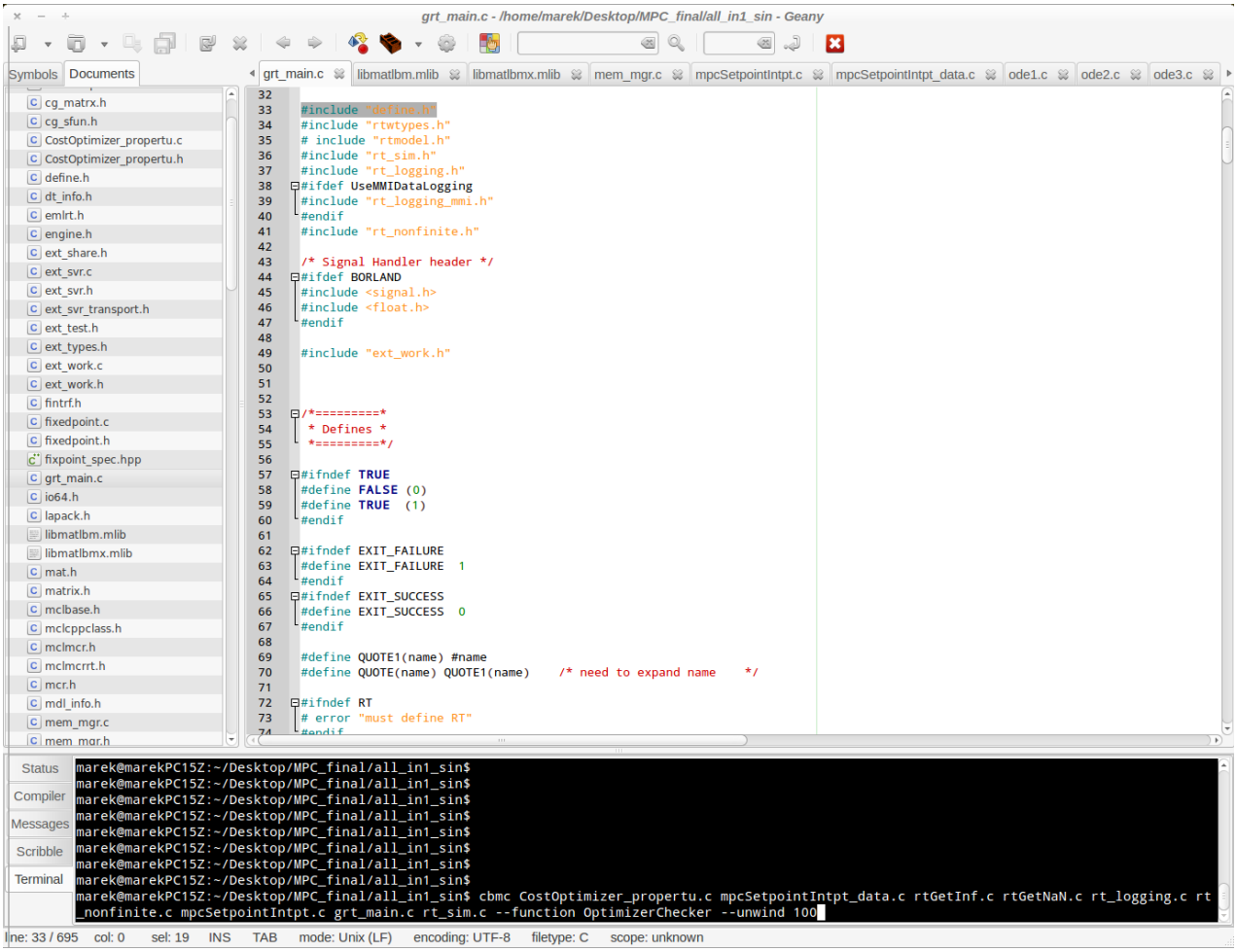


Fig.38.Screen shot of Geany, the light weight IDE utilised for compiling all the source code in one Linux platform

After running and compiling the source code, the C project generated the following object files:

- CostOptimizer_propertu.o
- grt_main.o
- mpcSetpointIntpt.o
- mpcSetpointIntpt_data.o
- rtGetInf.o
- rtGetNaN.o
- rt_logging.o
- rt_nonfinite.o
- rt_sim.o
- VerifyCostDecrease.o

Using the previous files, in terminal was commanded to create the executable file. Afterwards, when running this new executable, we obtained another *.mat file which was compared as it is shown below.

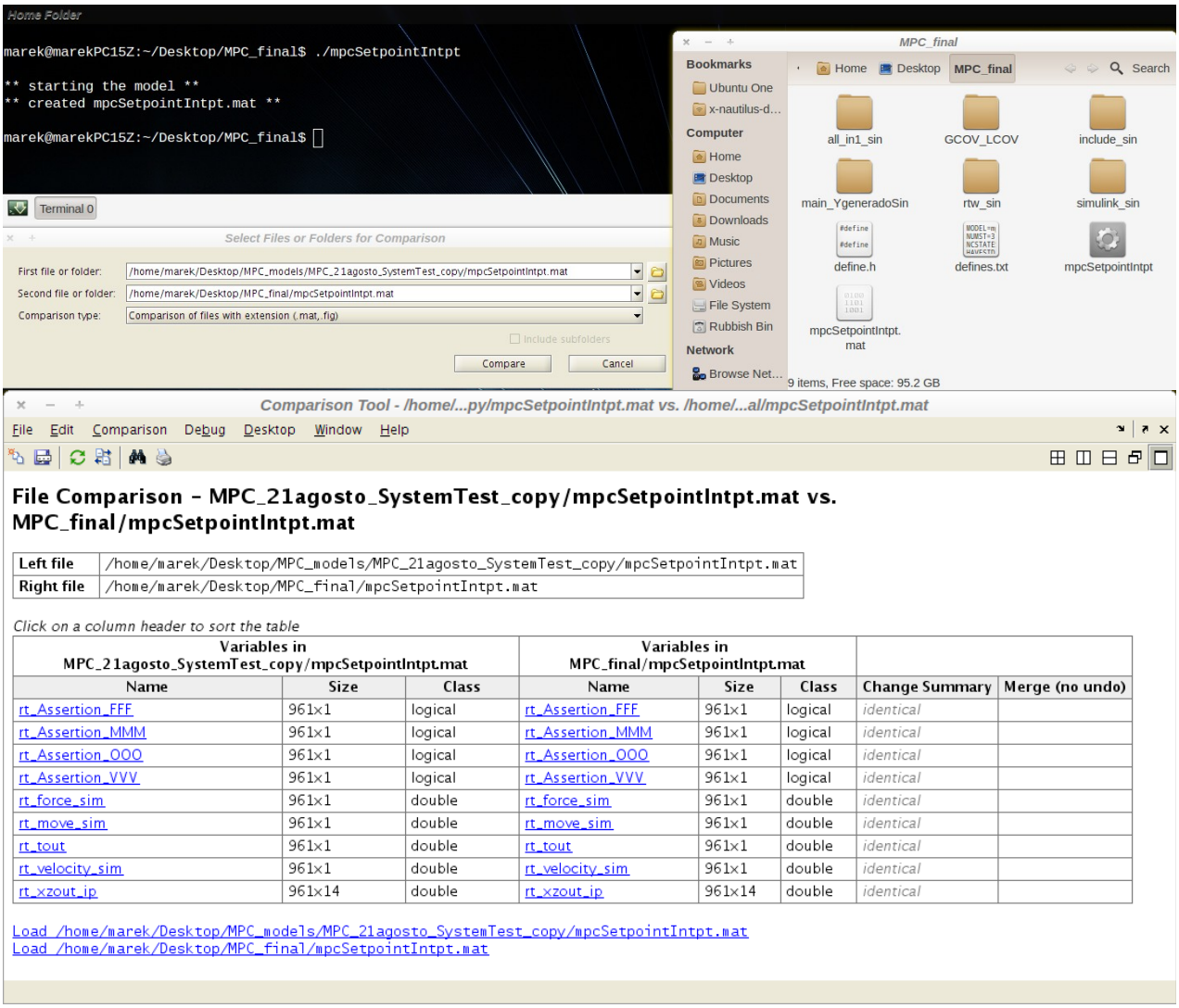


Fig.39.This screen shot shows clearly in one place, the command in terminal for the executable obtained in Geany (>>./mpcSetpointIntpt) and te directory, “MPC_final”. The window open below the terminal is the selector of Matlab for comparing *.mat files. After the selection of these two, the report indicates that all internal vectors for the control signal are identical.

5.3.- Analysis of Source Code

Once the C project solution was abstracted and its outcome was corroborated, the code was analysed. For this, *GCOV* was used to check the consistency of the code with regards to the coverage that could be obtained with this tool. This work was done mainly in terminal with command. To have a more graphical result, we employed *LCOV*. This latter tools gave us an overall coverage for the complete project.
For these we had to compile the source code, and employ additional flags. These are presented below.

```
>> gcc -c -ansi -pedantic -fwrapv -fprofile-arcs -ftest-coverage xxxx.c
```

The flags `-fprofile-arcs` and `-ftest-coverage` used in the compilation provided us with coverage analysis. Afterwards, the linking of the generated object files need to be linked using -fprofile-arcs once again.

```
>>gcc -fprofile-arcs -o ../mpcSetpointIntpt -lm
CostOptimizer_propertu.o mpcSetpointIntpt_data.o rtGetInf.o rtGetNaN.o
rt_logging.o rt_nonfinite.o mpcSetpointIntpt.o grt_main.o rt_sim.o
```

The gcov command was used to target *.c.gcov files obtained from the previous commands. *.gcda and *.gcno files were generated as well for each source code *xxxx*. The verbose outcome of this analysis can be seen in Appendix section E, but a clearer and more graphical result was obtained with LCOV. This tool used with the following command :

```
>>lcov --directory all_in1_sin --capture --output-file report.info
```

This command generated a .info file with the summary of the results from GCOV. But then to have this as a report we continue with the following instruction:

```
>> genhtml  report.info
```

This last command brings the test results into a HTML file that can be seen in the result section 7.
Information about the flags definitions and usage is presented in Appendix section F.

6 – Tracing assertions for CBMC

In this section we will describe how the assertion could be inserted manually in order to trigger the a mechanism that can be used by the bounded model checker.

6.1.- Assertions in the source code of C project

The assertion blocks of the Simulink model were translated and its usage in the code is based on another function-call which is *utAssert*. This assertion call are located in two different files: *mpcSetpointIntpt.c* and *CostOptimizer_propertu.c* .

6.1.1.-Optimizer Assertion in code

This block was translated into the file *CostOptimizer_propertu.c* . This appears in the code as follows:

```
/* Assertion: '<S1>/Assertion_OOO_ CostOptimizer' */
utAssert (rtb_LogicalOperator_o);
```

The utAssert takes a logical operator that is defined as a boolean_T. The reason why there is a separately file that describes the assertion set for the optimizer conditionals is that it was built in a atomic system. Remembering that there is a enable mechanisms, there was a need to make it run as a unit in one go during simulation. It is clear that the output-file *.mat contains the values of the assertion. But to use a BMC it was needed to simplify its target. When checking the code, there is a specific function-call that generates this output vector. This function is :

```
void mp_CostOptimizer_propertu_Start (RT_MODEL_mpcSetpointIntpt *const
  mpcSetpointIntpt_M, rtDW_CostOptimizer_propertu_mpc *localDW,
  rtP_CostOptimizer_propertu_mpcS *localP)
{
  /* Start for Enabled SubSystem: '<S1>/Cost_Comparator Subsystem' */
  /* InitializeConditions for UnitDelay: '<S10>/prevCost' */
  localDW->prevCost_DSTATE_b = localP->prevCost_X0;

  /* End of Start for SubSystem: '<S1>/Cost_Comparator Subsystem' */
  /* Start for SignalToWorkspace: '<S1>/Signal To Workspace' */
  {
    int_T dimensions[1] = { 1 };

    localDW->SignalToWorkspace_PWORK.LoggedData = rt_CreateLogVar(
      mpcSetpointIntpt_M->rtwLogInfo,
      0.0,
      rtmGetTFinal(mpcSetpointIntpt_M),
      mpcSetpointIntpt_M->Timing.stepSize0,
      (&rtmGetErrorStatus(mpcSetpointIntpt_M)),
      "Assertion_OOO",
      SS_BOOLEAN,
      0,
      0,
      0,
      1,
      1,
      dimensions,
      NO_LOGVALDIMS,
      (NULL),
      (NULL),
      0,
      1,
      0.25,
      1);
```

```
    if (localDW->SignalToWorkspace_PWORK.LoggedData == (NULL))
      return;
  }
}
```

The project works based on a state machine. That is the reason a simple function to generate values for an output signal has many arguments. The most efficient way to get a target for the BMC was to implement some manual replica of the logic operator and then assert them. This of course means a minor modification of the code but it does help the BMC to target a function-call which is an useful option provided by CBMC. Therefore, in the code it can be seen small definition of a function called *OptimizerChecker*, that only is referred when a the logic operator is used and then CBMC aims to get a counterexample to validate the provided assertion.

6.1.2.- Velocity, Force and Movement Assertions in code

The definition of these assertions are similar to that of the Optimizer. However, they are all implementation in the file *mpcSetpointIntpt.c*. They are all present inside the function:

```
void mpcSetpointIntpt_output0(int_T tid) /* Sample time: [0.0s, 0.0s] */
```

They have a similar structure of the optimizer definition and are described in the following order:

```
/* Assertion: '<Root>/Assertion_FFF_forces' */
        utAssert(rtb_LogicalOperator2);
/* Assertion: '<Root>/Assertion_MMM_moves' */
      utAssert(rtb_LogicalOperator);
/* Assertion: '<Root>/Assertion_VVV_velocity' */
      utAssert(rtb_LogicalOperator1);
```

For example, for the property block of the conditionals of the Force control we have:

```
rtb_LogicalOperator2 = ((mpcSetpointIntpt_B.Sum <=
mpcSetpointIntpt_P.Constant_Value_o) && (mpcSetpointIntpt_B.Sum >=
      mpcSetpointIntpt_P.Constant_Value_i));
```

Movement:

```
rtb_LogicalOperator   =   ((mpcSetpointIntpt_B.zOpt[0]   <=
mpcSetpointIntpt_P.Constant_Value_on)&&(mpcSetpointIntpt_B.zOpt[0]>=
      mpcSetpointIntpt_P.Constant_Value_d))
```

Velocity:

```
/* Constant: '<S23>/Constant' */
      mpcSetpointIntpt_B.Constant = mpcSetpointIntpt_P.Constant_Value_g;
/* RelationalOperator: '<S23>/Compare' */
      mpcSetpointIntpt_B.Compare = (mpcSetpointIntpt_B.Velocity <=
      mpcSetpointIntpt_B.Constant);
if (rtmIsMajorTimeStep(mpcSetpointIntpt_M)) {
/* Constant: '<S24>/Constant' */
      mpcSetpointIntpt_B.Constant_h= mpcSetpointIntpt_P.Constant_Value_b;
/* RelationalOperator: '<S24>/Compare' */
      mpcSetpointIntpt_B.Compare_g = (mpcSetpointIntpt_B.Velocity >=
      mpcSetpointIntpt_B.Constant_h);
if (rtmIsMajorTimeStep(mpcSetpointIntpt_M)) {
/* Logic: '<S6>/Logical Operator1' */
      rtb_LogicalOperator1 = ((mpcSetpointIntpt_B.Compare != 0) &&
      (mpcSetpointIntpt_B.Compare_g != 0));
```

6.2.- Code modifications for assertions

In order to be able to work with the BMC, we had to inform CBMC which function in the C source code we need to unwind and validate. It is clear that the code generated is not as simple as expected by an automatic checker. If CBMC is run without this information, it does run infinite unwinding of the complete program. Another point that we need to remember is the times violations took place in out initial test-bench set up in the high-level block model-Simulink. It was reported 27 property violations in the Optimizer constraints and 2 in the velocity property constrained.

6.2.1.- Optimizer Violations.

The code generated was studied and few arrangements were added in order to be able to deploy the BMC. In the file *CostOptimizer_propertu.c* .we added a few lines of code that basically call a function with two arguments. One of them is the value of the logical operator *var_opt*, and the other a counter *cnt_OOO* that carries information of the times violations happened in the Simulink model. As explained in section 2.2 CBMC will take the code and aim to create a boolean formulation from targeting a specific function-call. Thereafter, this tool will try to get a counterexample to falsify or validate the assertion inserted in the function.
The coded added is as follows:

```
var_opt=rtb_LogicalOperator_o;/*Added: extracting boolean values*/
 /*needed for checker argument marco */
if (var_opt == 0)
{cnt_OOO=cnt_OOO+1;}
OptimizerChecker (var_opt,cnt_OOO);  /*manually assert call */
```

And the function to be targeted by CBMC is :

```
/* ======================================================= */

void OptimizerChecker ( boolean_T var_opt, int cnt )
{
      if (cnt > 27)           /* Assertion: manually */
      {assert( var_opt  == 1 );}
       /*printf ("warning numb: %d \n", cnt);}*/


      /*  assert( rtb_LogicalOperator[0]  == 1 ) ; /* Assertion: manually */
      /*  assert( rtb_LogicalOperator[1]  == 1 ) ; /* Assertion: manually */
 }
/* ======================================================= */
```

Here by telling the BMC to assert the value of var_opt to be equal to 1 after the 27 errors occurs tries to find whether within the source code exists flows that could lead to an additional error. CBMC found these flows and provided a counterexample. For this to happen, the bounded model checker does not need initial values or states. It does find where the assertion that apparently will be always true, at least at high-level design, can have hidden errors in the low level model which is mainly used in the by hardware implementations. The outcome of this test is presented in section 7. The command used to target the function-call is:

```
>>cbmc  CostOptimizer_propertu.c  mpcSetpointIntpt_data.c  rtGetInf.c
rtGetNaN.c  rt_logging.c  rt_nonfinite.c  mpcSetpointIntpt.c  grt_main.c
rt_sim.c --function OptimizerChecker --unwind 100
```

6.2.2.- Physical Violations – Velocity property.

Considering the outcome of the test-bench implemented in section 4.1, we can see the times at which there were violations of properties. This is literally expressed in Appendix section B. Apart from the most recurrent errors from the optimizer, it was appreciated only two other violations with regards to the physical constraints. These two occurred at times 147.5 and 190.25 and came from the Velocity property block.
In this case implementation of assertion were similar to the one of the optimizer. Additionally, the absence of error being reported did not mean that the other property blocks (movement and forces control) were not checked. These apparent error free section were checked with the same approach. Assertion were inserted assuring that at all times no errors occur, or when no errors occur, the assertion was changed to use CBMC and see if the BMC could find a flow in code generated.
The coded added for the velocity property is as follows:

```
var_V=rtb_LogicalOperator1;/* Added: extracting boolean values */
  /*needed for checker argument marco */
      if (var_V == 0)
      {cntV= cntV+1;}
checkerVVV (var_V , cntV);        /*manually assert call */
```

And the function to be targeted by CBMC is :

```
/* ======================================================= */
/* Assert var_1, after the two error      */
/* this was seen in the Matlab Console    */

void checkerVVV (boolean_T var_1 , int cntV )
{
     if (cntV >2)

    { assert ( var_1  == 1 ) ;  /*  manual triggering */    }

   //assert( rtb_LogicalOperator[0]  == 1 ) ; /* Assertion: manually */
   //assert( rtb_LogicalOperator[1]  == 1 ) ; /* Assertion: manually */
     }
/* ======================================================= */
```

As seen above, the two velocity violations were jumped over in order to see if there is another flaw in apart from these errors. The bugs found can be seen in section 7. For the other two properties components, Force and Movement assertions were checked. The assertion aimed to prove that there is no error hidden in the code generated. With regards to the force control the code added was:

```
var_F = rtb_LogicalOperator2;
/* Added: extracting boolean values */

/*needed for checker argument marco */
checkerFFF (var_F);
/* manually assert call */
```

and its function-call for usage with the BMC is :

```
/* ======================================================== */
/* Assert var_1, is expected to fail       */
/* this was seen in the Matlab Console    */

void checkerFFF ( boolean_T var_1  )
{
     assert ( var_1  == 1 ) ;  /*  manual triggering */
//assert( rtb_LogicalOperator[0]  == 1 ) ; /* Assertion: manually */
//assert( rtb_LogicalOperator[1]  == 1 ) ; /* Assertion: manually */
}
/* ======================================================== */
```

And the Movement control block included that following code:

```
var_M = rtb_LogicalOperator;
/* Added: extracting boolean values */
/* needed for checker argument marco */
checkerMMM (var_M);
/*manually assert call */
```

and its function-call:

```
/* ======================================================== */
/* Assert var_1, is expected to fail       */
/* this was seen in the Matlab Console     */
void checkerMMM (boolean_T var_1     )
{
     assert ( var_1  == 1 ) ;  /*  manual triggering */
//assert( rtb_LogicalOperator[0]  == 1 ) ; /* Assertion: manually */
//assert( rtb_LogicalOperator[1]  == 1 ) ; /* Assertion: manually */
}
/* ======================================================== */
```

6.3.- CBMC target

Once the target functions were implemented, they were called individually. This manual triggering of assertion was possible and practical for our analysis. Asserting *var_1* in the constrained property controller to be equal to 1. Meaning that it was tried to prove that there was a possible state condition chain that could produce an outcome violating safety-critical properties. These function-call were targeted by CBMC using the following commands:

- For force control:

```
>>cbmc   CostOptimizer_propertu.c   mpcSetpointIntpt_data.c   rtGetInf.c
rtGetNaN.c  rt_logging.c  rt_nonfinite.c  mpcSetpointIntpt.c  grt_main.c
rt_sim.c --function checkerFFF --unwind 100
```

- For movement control:

```
>>cbmc   CostOptimizer_propertu.c   mpcSetpointIntpt_data.c   rtGetInf.c
rtGetNaN.c  rt_logging.c  rt_nonfinite.c  mpcSetpointIntpt.c  grt_main.c
rt_sim.c --function checkerMMM --unwind 100
```

- For velocity control:

```
>>cbmc CostOptimizer_propertu.c mpcSetpointIntpt_data.c rtGetInf.c
rtGetNaN.c rt_logging.c rt_nonfinite.c mpcSetpointIntpt.c grt_main.c
rt_sim.c --function checkerVVV --unwind 100
```

- For the optimizer control:

```
>>cbmc CostOptimizer_propertu.c mpcSetpointIntpt_data.c rtGetInf.c
rtGetNaN.c rt_logging.c rt_nonfinite.c mpcSetpointIntpt.c grt_main.c
rt_sim.c --function OptimizerChecker --unwind 100
```

In order to be able to analyse the counterexample provided by CBMC, script commands were used to save the information of the states. Additionally, CBMC allowed us to report the counterexample in a XML format. This is done by adding the following flag : `--xml-ui` .
So as an example for us to have the results saved for the velocity control, the following command was used(provided that location of directory is *"all_in1_sin"*):

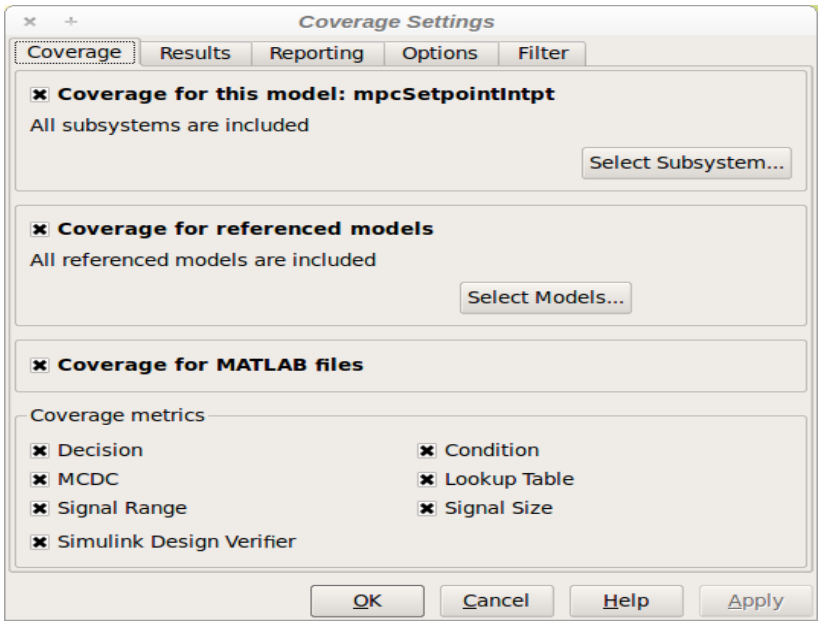
```
>> script report_velocity.txt
>>cbmc CostOptimizer_propertu.c mpcSetpointIntpt_data.c rtGetInf.c
rtGetNaN.c rt_logging.c rt_nonfinite.c mpcSetpointIntpt.c grt_main.c
rt_sim.c --function checkerVVV --unwind 100 --xml-ui

>>[ctrl+D]
```

Afterwards, the counterexample is given in the *report_velocity.txt.* This file needs some modification. It was needed to delete all terminal symbols so that it could be opened by any browser. After that, the file format was changed from *.txt to *.xml. Reading the outcome from xml file in a browser of not of much help. For this, it was employed an especial xml reader called BASEX (v.7.0.2). BASEX is a scalable xml databased viewer that has many functionalities useful for visualization of large structural reports. This report format is presented and explained in section 7.

7 – Results

7.1.- Block-model Consistency and Coverage with V&V Simulink



In this section we present the the result of the block model design under test. The high-level model was set up in Simulink and by using the Verification and Validation toolbox . The coverage setting (Tools > Coverage Settings) is show in the Figure on the left. The block model is clear to understand and very practical when it is needed for testing of new design and development of controllers, in this case a model predictive control.
The final report obtained is shown below:

Fig.40. Here is show the features available for coverage provided by V&V toolbox.

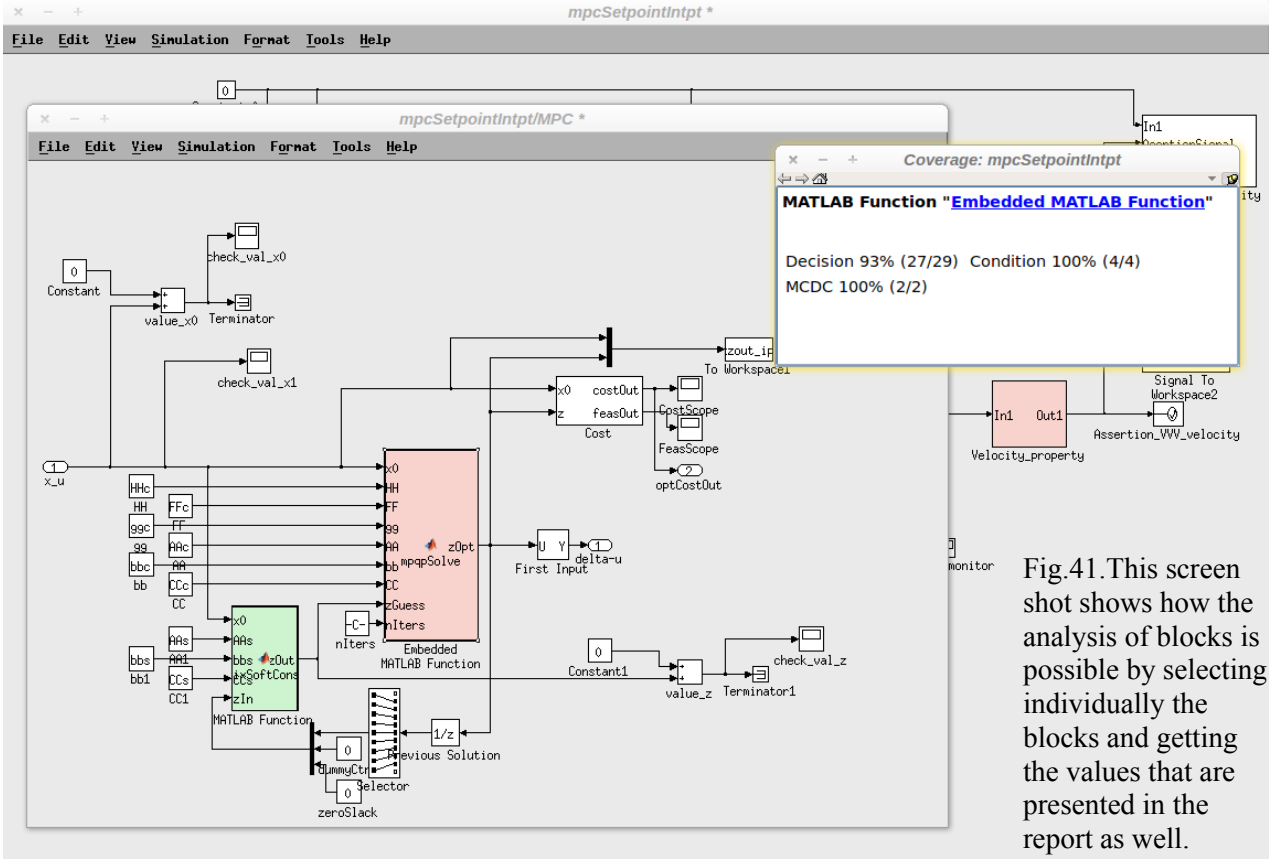


Fig.41.This screen shot shows how the analysis of blocks is possible by selecting individually the blocks and getting the values that are presented in the report as well.

The colouring of the model blocks represent completion the coverage. This exposes the coverage results in a diagram for the path that has been covered. Green block represents a 100% coverage, red path had a partial coverage and the grey had none (node representation).

The coverage report of the block-model is as follows:

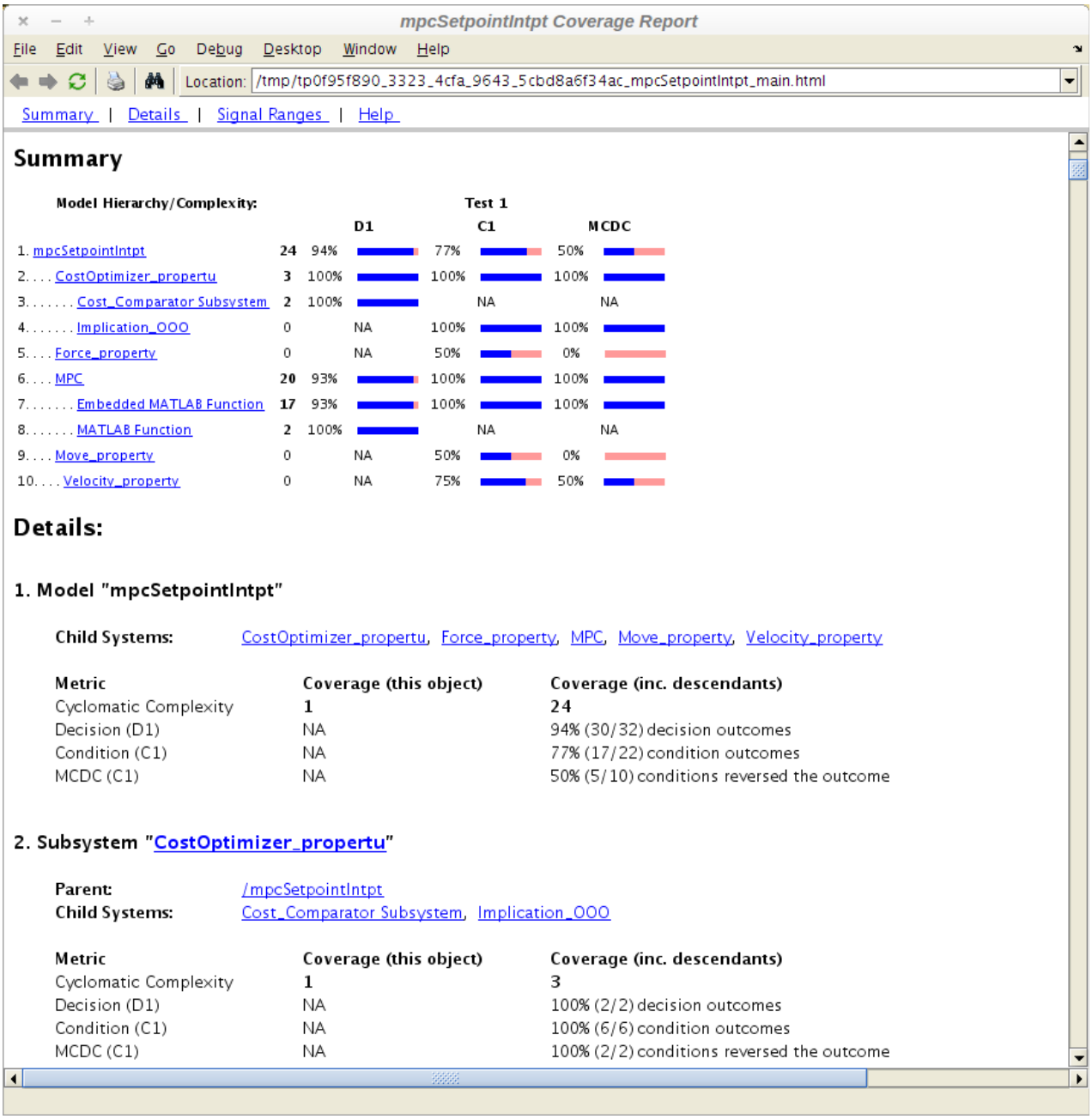


Fig.42.Summary of the coverage report. This include a vast information with regards of variables/blocks that were covered and the inputs that were never tested creating the non exercised regions in the model design.

The coverage of the MPC set up in Simulink has promising values. The areas not sufficiently exercised could be improved by extending the simulation time, or modifying the constraints to be more critical and bring up properties violations on purpose to in order to cover the remainder. This may incur in changing some features of the MPC design. The column D1, C1 and MCDC corresponds to Decision, Conditional and Modified condition/decision coverage respectively. Definitions of these coverage metrics are: Decision coverage (also called Branch coverage) has control structure that depends on conditional such as "if" statements being evaluated both to be true or false. Condition coverage has boolean statements being evaluated both to be true and false. This does not necessarily means branch coverage. Modified Condition / Decision Coverage (MC/DC) has conditional statements in a decision taken on all possible outcomes that could be present at least once during simulation. MC/DC has statements showing how they individually affect that decision outcome.

7.2.- Code Consistency and Coverage with GCOV and LCOV

Once the MPC design was tested, it went through a translation process that was done by the code generation of the high-level model. After the modifications explained in section 5, the project built with the C source code was tested with GCOV and then LCOV. The latter provided us with the following results:

LCOV - code coverage report

		Hit	Total	Coverage
Current view: top level - GCOV_LCOV/all_in1_sin				
Test: app.info	Lines:	1237	2665	46.4 %
Date: 2012-09-05	Functions:	70	83	84.3 %
	Branches:	452	1353	33.4 %

Filename	Line Coverage		Functions		Branches	
CostOptimizer_propertu.c	100.0 %	34 / 34	100.0 %	5 / 5	90.0 %	9 / 10
grt_main.c	48.9 %	66 / 135	66.7 %	2 / 3	45.2 %	38 / 84
mpcSetpointIntpt.c	94.7 %	680 / 718	96.8 %	30 / 31	81.7 %	227 / 278
rtGetInf.c	78.9 %	30 / 38	100.0 %	4 / 4	40.0 %	4 / 10
rtGetNaN.c	80.0 %	20 / 25	100.0 %	2 / 2	37.5 %	3 / 8
rt_logging.c	23.1 %	391 / 1694	71.0 %	22 / 31	17.6 %	167 / 949
rt_nonfinite.c	75.0 %	12 / 16	60.0 %	3 / 5	25.0 %	3 / 12
rt_sim.c	80.0 %	4 / 5	100.0 %	2 / 2	50.0 %	1 / 2

Generated by: LCOV version 1.9

Fig.43.Report of the coverage for source code abtracted

As it is shown in the Figure above, the code coverage brings up some results that are not as high as the one we obtained in the high-level model using blocks in Simulink. It is clear that the code generating toolbox is not that optimal, but still these results were expected since the description of a complex model in a low level language is likely to have more flows.

LCOV - code coverage report

		Hit	Total	Coverage
Current view: top level				
Test: app.info	Lines:	1237	2665	46.4 %
Date: 2012-09-05	Functions:	70	83	84.3 %
	Branches:	452	1353	33.4 %

Directory	Line Coverage		Functions		Branches	
GCOV_LCOV/all_in1_sin	46.4 %	1237 / 2665	84.3 %	70 / 83	33.4 %	452 / 1353

Generated by: LCOV version 1.9

Fig.44. Summary of LCOV Coverage report

The code coverage for the top level 46,4 %. This is definitely a low value that may be the cause of error that are not present during simulation. However, once implemented in hardware and with the insertions of external disturbances, this areas that were not either covered or exercised represents a great danger in when verification and then validation is attempted.

7.3.- CBMC reports

After studying the BMC tool and options, the tracing of assertions was possible. Its usage can be extended to check specific areas not covered in the previous tests. As a case in point, for the verification and validation of our model, we need to ensure correctness of the model design even for the code areas not covered (i.e. code coverage). That is why using CBMC, it was possible to check these non - exercised code blocks. The assertions and its correspondent function-calls aim to show if it is possible to get an

counterexample that falsifies a conditional that represents a high-level constraint. The result of these tests are presented in this order : Optimizer control with function-call *OptimizerChecker*, Force control with function-call *checkerFFF*, Movement control with function-call *checkerMMM* and Velocity control with function-call *checkerVVV*.

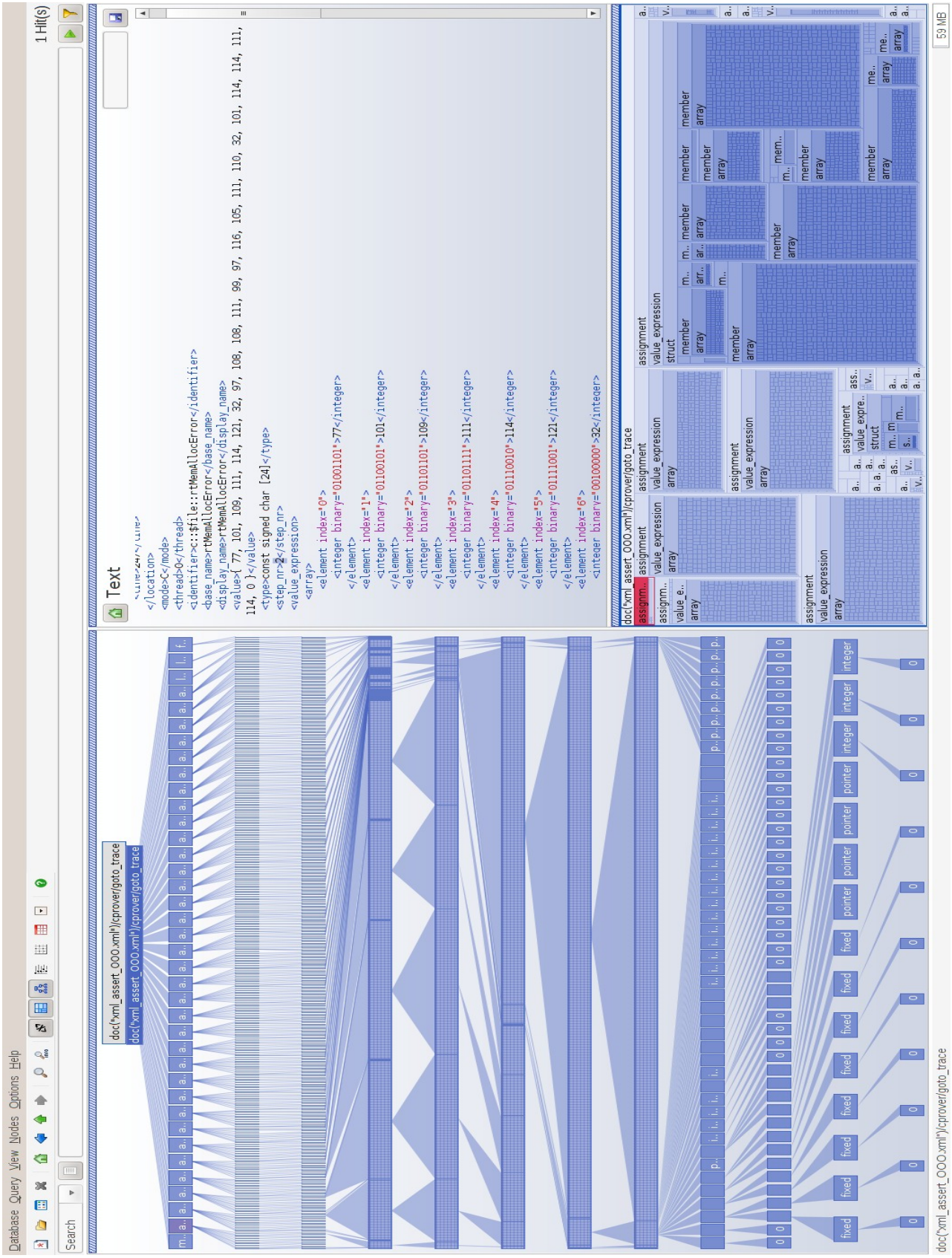


Fig.45. XML Report for the counterexample targeting the *OptimizerChecker* provided by CBMC.

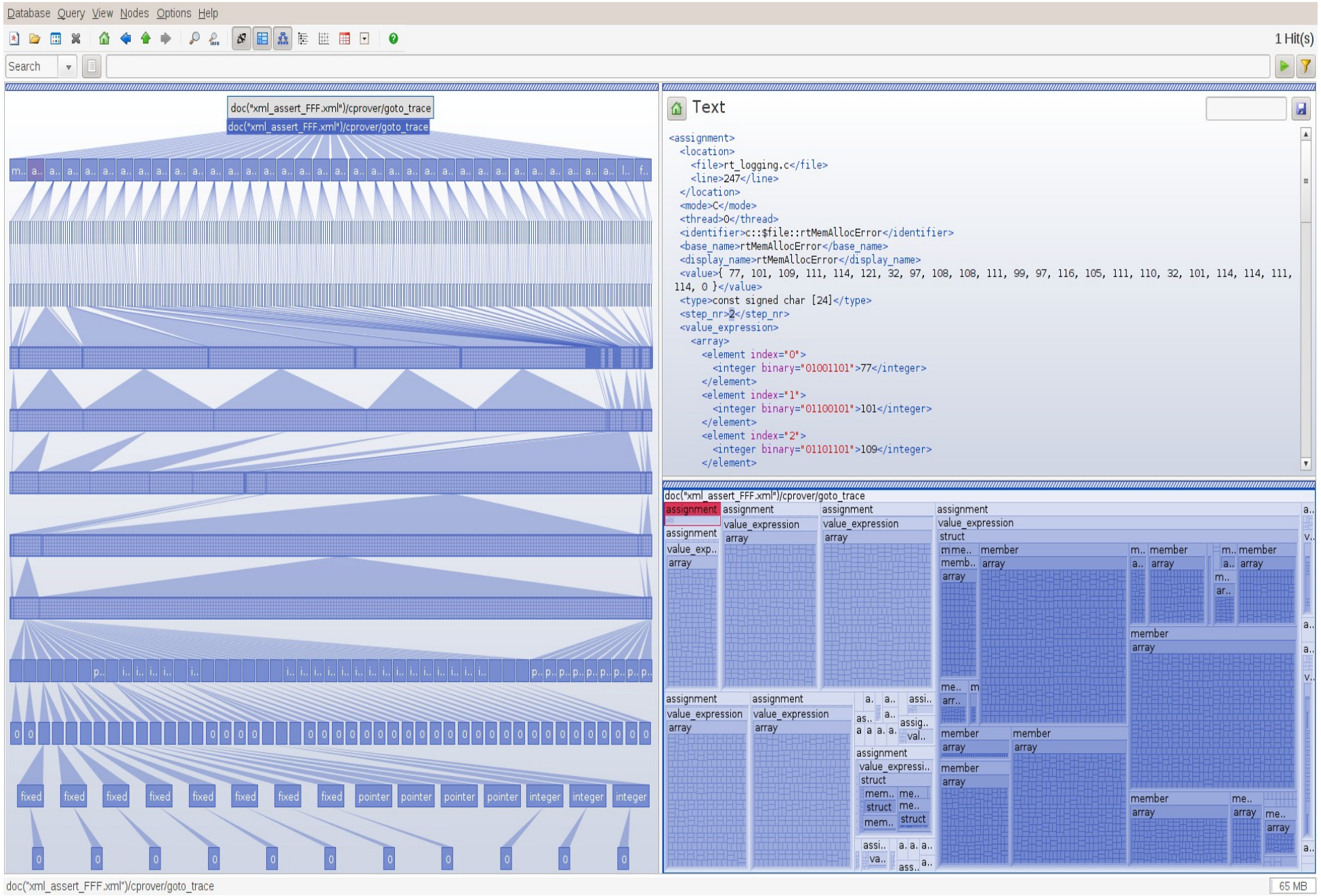


Fig.46.XML Report for the counterexample targeting the *checkerFFF* provided by CBMC.

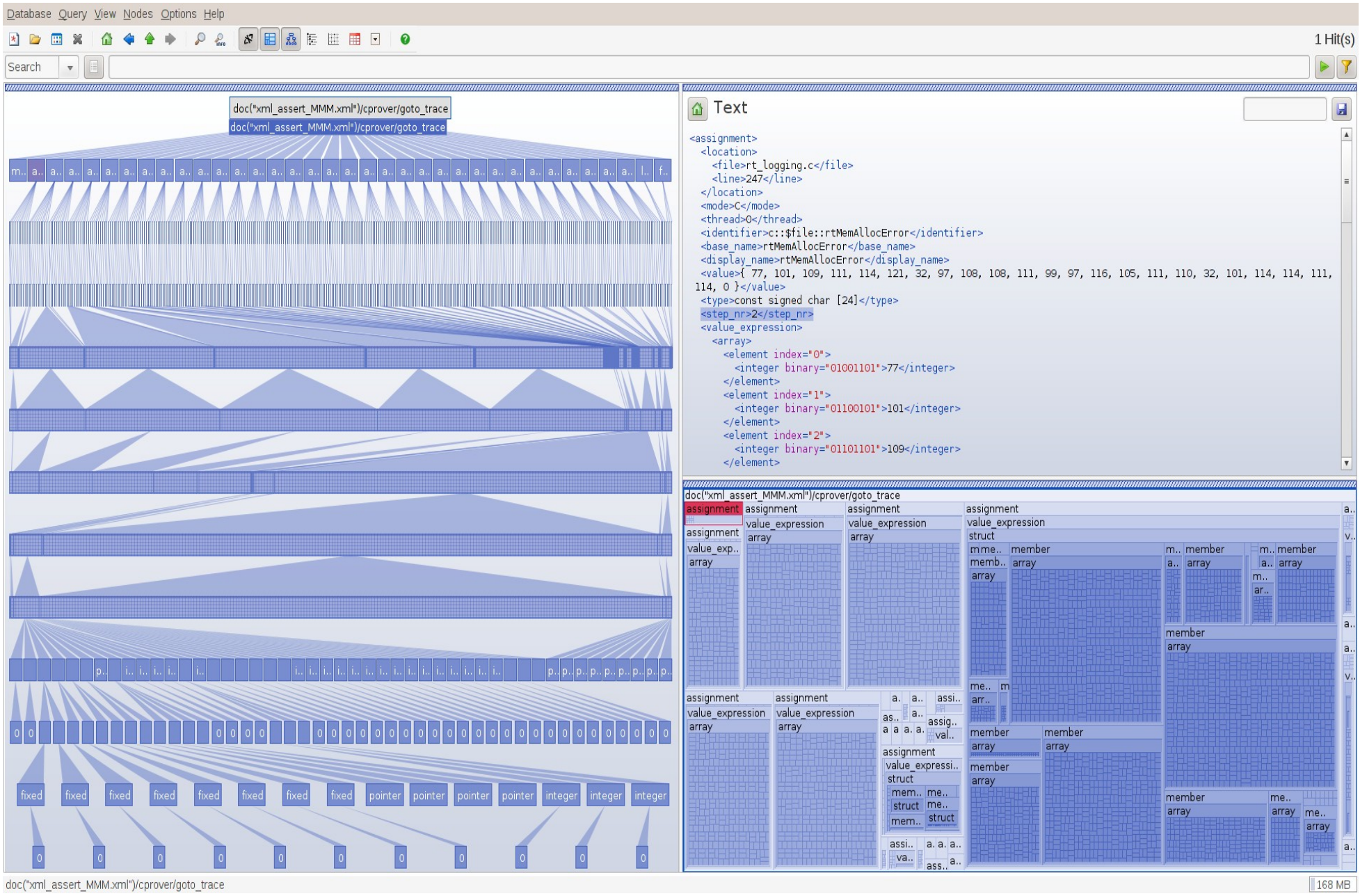


Fig.47.XML Report for the counterexample targeting the *checkerMMM* provided by CBMC.

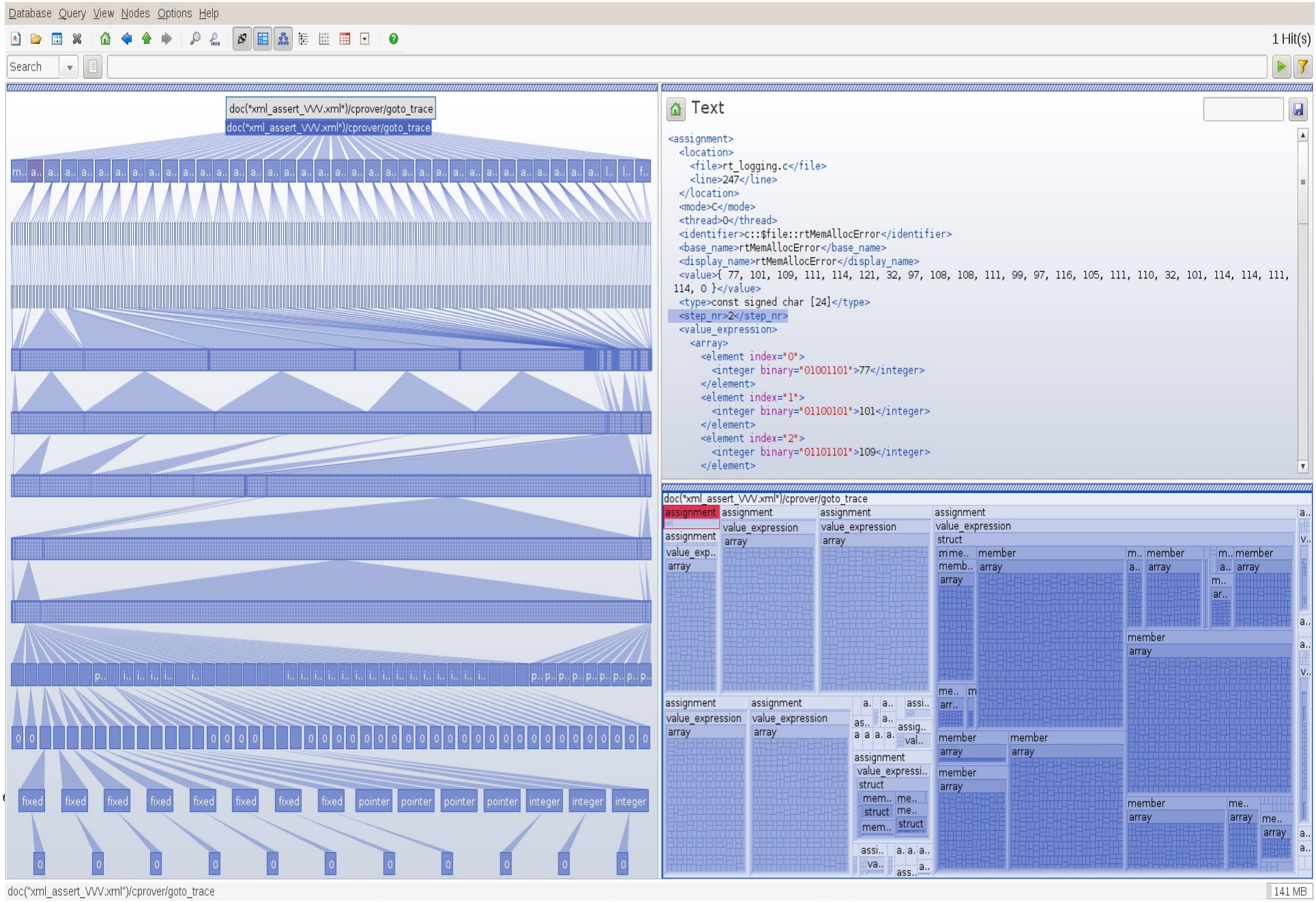


Fig.48.XML Report for the counterexample targeting the *checkerVVV* provided by CBMC.

In the previous figures (last 4), it is shown how CBMC managed to point out bugs within the C source code of the model predictive control. On the left hand side BaseX viewer panel, we can see a tree visualization of the structural report provided by the BMC. On the right, we see the xml text over a map visualization. This is a practical way to see how the states are built and which inputs this had. If we go to a deeper or lower hierarchy so that we can observe how the bounded model checker constructed its counterexample. The lower we go in the tree, the more specific the it becomes with regards of program (type of variables, memory values, etc.). For example is we decide to check the branch *<step_nr>30</step_nr>* in the xml report for the optimizer, we will see the following tree.

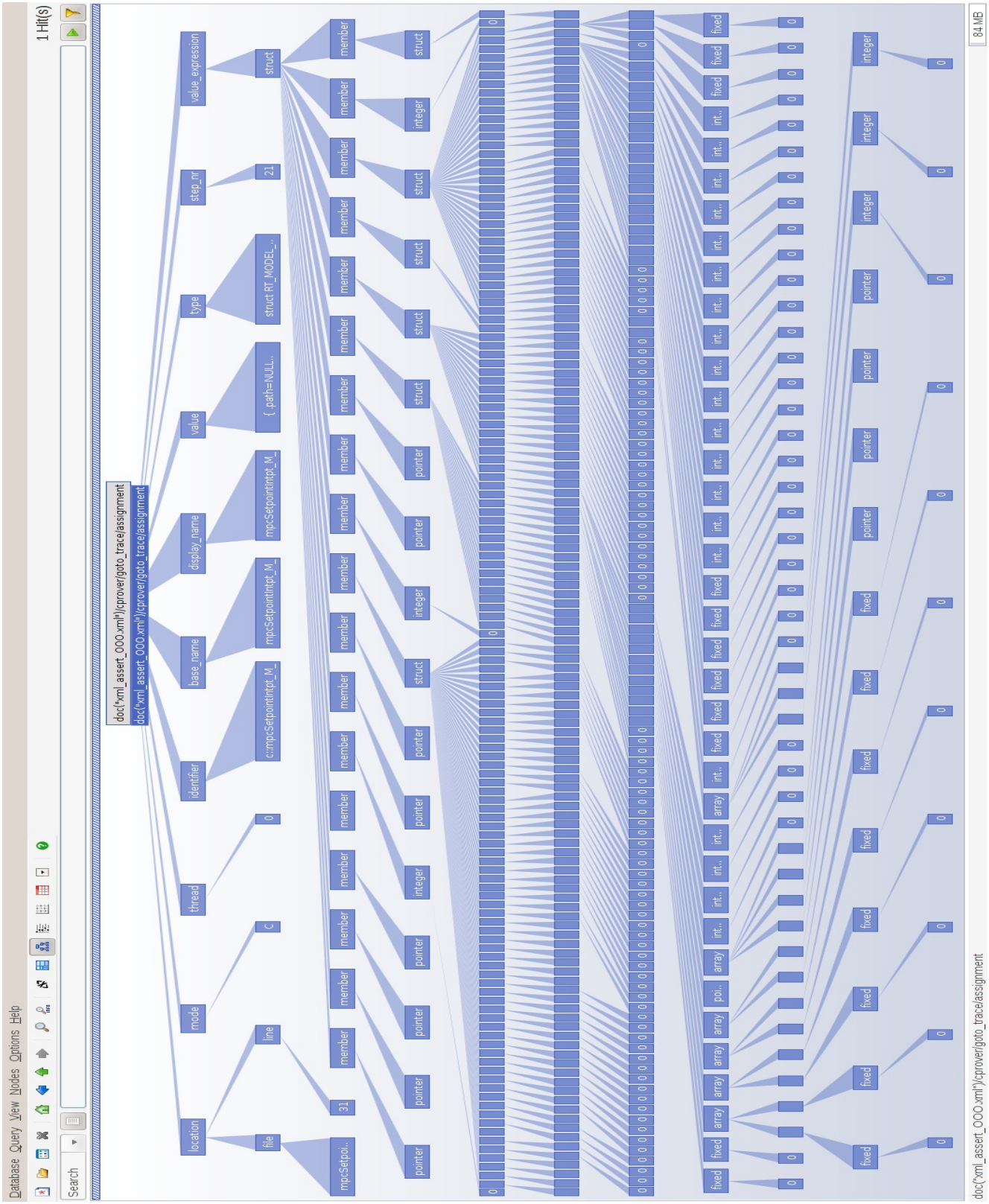


Fig.49. Analysis of the states in counterexample in a deeper hierarchy

8 – Discussion

This section will discuss the obtained results along the implementation and research that took place in this project and explain whether or not research and techniques questions have been answered. It will compared the results provided by the dual analysis of the advanced predictive control strategy. Additionally a appropriate review of the aims and objectives will be explained whether they have been completed and where further work could take place.

8.1.- Choice of Analysis

The complexity and the logic undecidability presented in the MPC algorithm makes its verification and validation a challenge. Hence, this project was concentrated in exploring method(s) in order to verify this algorithms employing techniques and tools commonly used in the microelectronic and computer science industry.

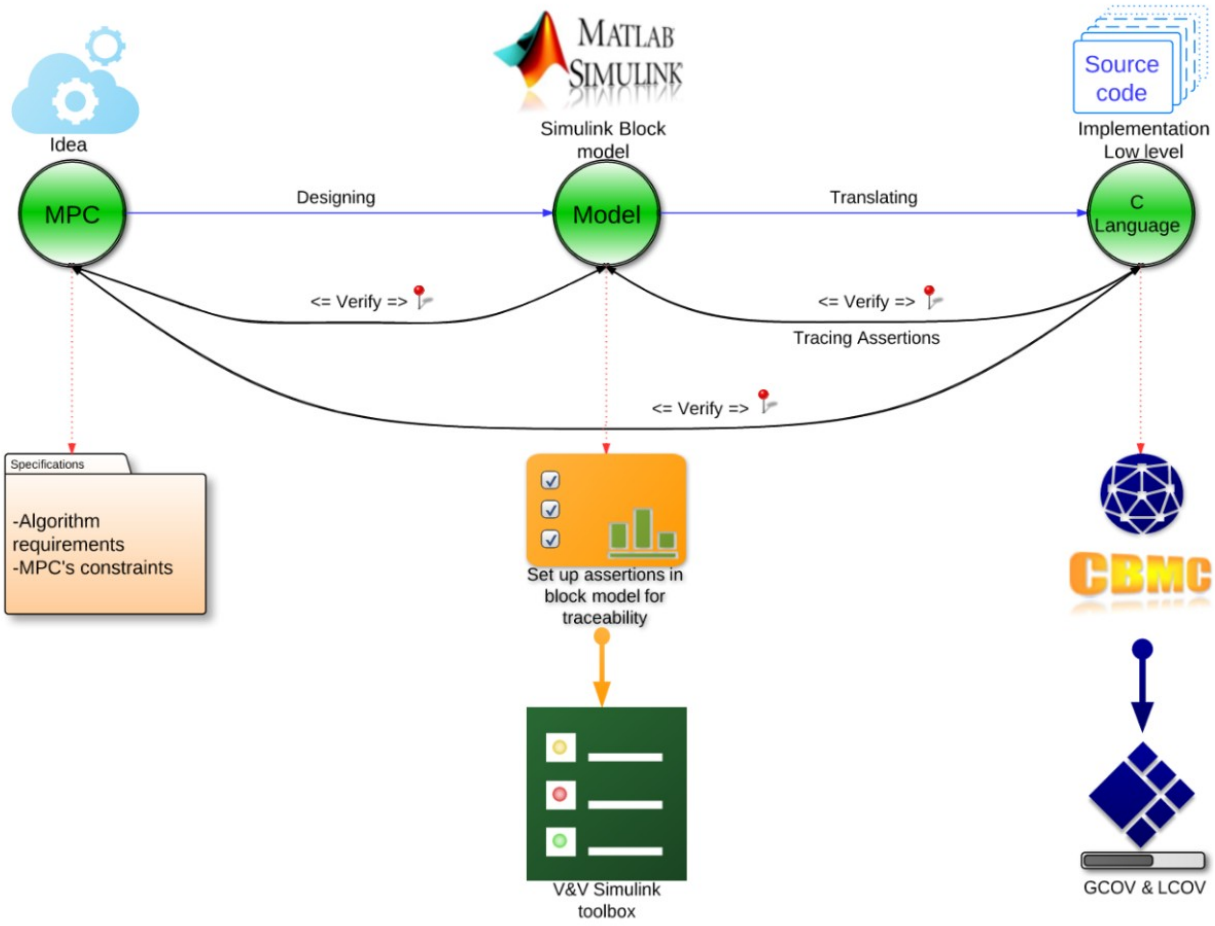


Fig.50. Diagram of the method employed for the verification task.

The Figure above summarises the approach taken in this project. It is a novel way to stablish a verification environment. Simulink was employed, a well know tool for controllers design and then with its code generator toolbox, it was possible to build a C project/solution in MSVS. After ensuring that the generated code was correct, a bounded model checker - formal verification tool- was used to trace the errors and find other bugs hidden in the code. The algorithm was checked in high and low level model description and coverage was obtained in both cases. During the research period, there were other similar tools that could have been used. In the case of the bounded model checker, FRAMA-C seemed to have good amount of features but due the time constraints for this project it could not be tested with the final C source code, only small examples showed its capabilities. On the other hand, Matlab has multiple toolboxes that could have improve the analysis of the algorithm and the traceability of the model block into the code generated. One

of these is Polyspace Model Link™ SL which is a static analyser. SystemTest™ also could have helped us to set up test-benches for different parts of the block model. Due to a problem with licensing and time limitations, it was not possible to use these tools to analyse our model.
The starting study undertaken to understand the basic of a model predictive control was important since it gave us a better view of the model and also showed us where and how verification method(s) could be implemented more intelligently. This was also crucial to state the assertion base on the specification of the model design.
The bifurcated approach aimed to set a verification environment in order to be able to employ well known techniques in verification and validation field. As it was shown in the previous section, the results of testing the block-model in Simulink with V&V toolbox are promising. The main reason of this is the high coverage obtained. However, for us to have a certification approval for a safety-critical software, we need reassurance of the MPC implementation at a lower level plus a overall coverage of higher range. This analysis was then transferred into a C language, describing the operability and functionality of the model. It was here were by using a formal verification tool (CBMC), it was aim to trace the properties of the high-level model. These were interpreted as assertion in the code so that CBMC could be deployed and in search of bugs hidden in the source code of the C project or solution called in MSVS. The results of the coverage and consistency of the code generated were lower as expected. This principally happened because of two reasons: Firstly, the code generation toolbox is not optimal with regards of its structural consistency. Secondly, the translation of a model from a higher description environment into a lower one will always include factors that were not acknowledged in the high-level model. CBMC proved to be a powerful tool in bug-hunting. It was possible to corroborate the property violations presented in the high-level model but even more, showed us that this approach is favourable to reach these uncovered areas exposed with GCOV and LCOV. The bounded model checker managed to find errors that come from the code generated project and stablish a new way to catch these in non exercised code blocks or lines.
The clear visualization provided by CBMC report would mean an increase of efficiency in catching bugs with complex predictive controls designs. Furthermore, this may easily lead into modifications for improvements in the MPC model set up in Simulink. Tracing back these into a higher model could surely have helped us find bugs/errors within the high-level model description and even Simulink toolboxes. Therefore, the choice of analysis to tackle this verification task is a flexible one given that the same model is studied at different levels and it functionality and boundaries were checked in order to corroborate its correctness.

8.2.- Evaluation

The first objective of this project was the study of the model predictive control strategy to establish the constraints for a particular controller. This was completed with the theoretical explanation given in the section 2 of background and it was used in the implementation of property blocks set in the Simulink model description for MPC. Apart from the physical boundaries for correctness of the predictive controller functionality, the optimizer was also studied and an assertion was set to prove whether or not it does work properly. The second and core objective of this project was for sure the exploration of methods that could lead into a reliable verification and validation process for MPC designs. This study concluded in the selection of a bounded model checker known in the microelectronic and computer science engineering fields. This second objective was completed in stages. The first one did successfully set the boundaries for MPC design in Simulink. The second stage was focused on the code abstraction using a Microsoft Visual Studio and Matlab's Code Generation toolbox. This was confirmed and just then it was possible to continue with the initial plan of the verification environment to be used. The third stage was completed by exposing how a formal tool in verification can work in the description model of a MPC. This third stage arose more issues to be considered in verification phase but also in the design phase. Here, special focus was set to analyse the way the code generator works. This was done mainly to be able to trace the block-properties. Once this was done, there were modification that enabled the CBMC to target manually inserted assertion that were consistent with the boundaries of the MPC's specifications. In other words, in this third stage the verification and validation “cycle” was set up for a advanced controller. This cyclic development is commonly seen when designing a new microchip or software package. Going from prototype to test-bench through debugging process for the MPC , it was achieved a new verification scheme for the particular controller.

In addition to the previous two objectives, a final coverage report was aimed to compare the coverage result of both fronts of analysis. This was completed by using the V&V Simulink toolbox for the block model and LCOV and GCOV for the generated C code. As a way to formulate a comparison, these two coverage results were obtained to have them as reference while answering the why CBMC was able to get error not found previously.
This project dealt with the decision making involved in the verification and validation of an advanced predictive control strategy. It included the study of theoretical background for understanding of the problem, the tools employed and the outcome of different testing used for the debugging of the MPC model. Ultimately, it exposes novel techniques that can be used for verification of complex systems that are difficult to be formulated due to its versatility. Moreover, it also proved that there are ways in which well known techniques in verification, such as bounded model checking can be used to debug a predictive control design.

8.3.- Future Work

The framework employed along this project is amenable to changes and improvements. Hence, the methods that took place could be performed using other tools improving efficiency (i.e. time and cost).
For example, during the code abstraction, the toolbox PolySpace would have been of great help since it has features that help the traceability of code generated. Along the usage of CBMC, we could have employ other model checkers (e.g. Frama-C, LLBMC,etc.).Or even inside this automatic model checkers, we could have decided to employ a SMP solver (Satisfiability Modulo Theories) instead of a SAT solver. It is evident that the available code analysis/consistency testing tools for C language are more numerous than the ones available for the block model. Ensuring the correctness of this bounded model checker with multiple tools would be of great advantage for a verification approval.

With regards to the theoretical part, the hard and soft constraints presented in the predictive model could have been broken down during its study and checked for flaws individually. It was crucial for this project the study of other model predictive control models in order to find a best fit solution to formalize specification properties. This process of formalizing specification is of great interest because it allows a more mathematical verification by the use of lattice[33] as a mean to prove algorithms. However, this would only represent a high-level modelling verification.
There is a number of studies [28,29,30] centred in the validation and verification of systems while using code generator. These were studied but it is clear that there is an special attention towards the development of new verification environments. Then, depending on the nature of the algorithm employed , it could be designed an improved verification model. This may be an hybrid model. This new verification environment or hybrid verification model would be shaped in answer to the complexity brought by MPC strategy. And also by the effects caused by translating the model from a high-modelling block-model to a low-level language.
Once mentioning this latter factor, an study of additional code generation tools available would give us more ground to work on further testing. Automatic code generation could be used as a powerful tool in verification. This promising idea of having a tool that codes automatically what is designed and tested in a high-level is well accepted in many industries, but not in controllers with safety-critical limitations. For example, the Code Generator toolbox from Simulink has the option of generating HDL(hardware description language) code. By doing this, it will be necessary to check the correctness of the translation and then the V&V will look more like a classic verification task in microelectronics.

During this research project, it was developed a new way to tackle the task of verifying a model predictive controller. The usage of further tools that could not be put in practice but they may have brought more result and helped us to undertake a more comprehensive V&V procedure.
This project definitely opens a new perspective that enhances the chances and liability of verification engineers to solve a problem that has been an obstacle for MPC design to be deployed widely and developed in areas such as aeronautics.

9 – References

1. Arthur Richards, William Stewart and Alex Wilkinson , Auto-coding Implementation of Model Predictive Control with Application to Flight Control, Proceeding of the European Control Conference, No. AIAA-2009-5780, August 2009.
2. Pontus Bostrom, Contract-Based Verification of Simulink Models, Abo Academi University, Turku, Finland.
3. James P. Keithan, David Landoll , Paul Marriott, Bill Logan, The Use of Advanced Verification Methods to Address DO-254(hardware) Design Assurance, Aerospace Conference 2008 IEEE, Montana, USA.
4. Kristoffer Karlsson and Håkan Forsberg, Emerging Verification Methods for Complex Hardware in Avionics, Saab Avitronics, Jönköping, Sweden.
5. N. M. C. de Oliveira and L. T. Biegler, "Constraint handling and stability properties of model predictive control," American Institute of Chemical Engineers' Journal, vol. 40, p. 1138–1155, 1994.
6. Arthur Richards, Smart Loitering for UAV Collision Avoidance, Department of Aerospace Engineering, University of Bristol, UK.
7. Control Chi-Ying Lin, and Yen-Chung Liu, Precision Tracking Control and Constraint Handling of Mechatronic Servo Systems Using Model Predictive.
8. Ian Dodd, Ibrahim Habli. Safety Certification of airborne software: An empirical study, Reliability Engineering and Safety, Vol. 98, p.7-23. DOI link: http://dx.doi.org/10.1016/j.bbr.2011.03.031
9. RTCA: DO-178B, "Software considerations in Airborne Systems and Equipment Certification," Washington, D.C., USA , 1992.
10. Martin S. Feather, Lorraine M. Fesq, Michael D. Inhgam, Suzanne L. Klein, Planning for V&V of the Mars Science Laboratory rover software, 2004 IEEE Aerospace Conference Proceedings, Chicago, USA.
11. Mathworks Inc.: Simulink , http://www.mathworks.com
12. C. Cutler and B. Ramaker. Dynamic matrix control-a computer control algorithm. American Institute of Chemical Engineers, Houston, Texas, 1979.
13. T. Tsang and D. Clarke. Generalized predictive control with input constraints. IEE PROCEEDINGS, Part D., 135:415–460, 1988.
14. J. Rawlings and R. Muske. The stability of constrained receding horizon control. IEEE Transaction in automatica Control, 38:1512–1516, 1993.
15. M.Kothare, V. Balakrishnan, and M. Morari. Robust constrained model predictive control using linear matrix inequalities. Automatica, 32, 1996.
16. P. Scokaert and D. Mayne, "Min-max model predictive control for constrained linear systems," IEEE Transactions on Automatic Control, vol. Vol 43 No 8, pp. 1136–1142, August 1998.
17. L. Chisci, J. A. Rossiter, and G. Zappa, "Systems with persistent disturbances: Predictive control with restrictive constraints," Automatica,vol. 37(7), pp. 1019–1028, 2001.
18. P. J. Goulart, E. C. Kerrigan, and J. M. Maciejowski, "Optimization over state feedback policies for robust control with constraints," Automatica, vol. 42, pp. 523–533, 2006.
19. A. G. Richards and J. P. How, "Robust stable model predictive control with constraint tightening,"

in Proceedings of American Control Conference, Minneapolis, Minnesota, 2006, pp. 1557–1562.

20. J. Maciejowski, Predictive Control with Constraints. Prentice Hall, 2002.

21. Kalman, R. (1960b). A new approach to linear filtering and prediction problems. Transactions of ASME, Journal of Basic Engineering, 87, 35–45.
22. Richalet, J., Rault, A., Testud, J. L., & Papon, J. (1978). Model predictive heuristic control: Applications to industrial processes. Automatica, 14, 413–428
23. Cutler, C., Morshedi, A., & Haydel, J. An industrial perspective on advanced control. In AICHE annual meeting, Washington, DC, October 1983.
24. Bemporad, A., Morari, M., Dua, V. and Pistikopoulos, The explicit linear quadratic regulator for constrained systems. Automatica 38 1, pp. 3–20, 2002
25. Qin, SJ., Badgwell, TA., A survey of industrial model predictive control technology, Control Engineering practice 11 : 733-764, 2003
26. Davide Bresolin, Luigi Di Guglielmo, Luca Geretti, and Tiziano Villa Correct-by-Construction Code Generation from Hybrid Automata SpecificationIn the proceedings of IEEE Workshop on Design, Modeling and Evaluation of Cyber Physical Systems (CyPhy) Istanbul, Turkey , July 5-8, 2011
27. Jonas Witt and Herbert Werner (2010). Approximate Model Predictive Control for Nonlinear Multivariable Systems, Model Predictive Control, Tao Zheng (Ed.)

28. Giuseppe Di Guglielmo, Luigi Di Guglielmo, Franco Fummi, and Graziano Pravadelli *Enabling Dynamic Assertion-based Verification of Embedded Software through Model-driven Design* In the proceedings of ACM/IEEE Design, Automation and Test in Europe, Dresden, Germany, 2012

29. Davide Bresolin, Luigi Di Guglielmo, Luca Geretti, and Tiziano Villa *Correct-by-Construction Code Generation from Hybrid Automata Specification* In the proceedings of IEEE Workshop on Design, Modeling and Evaluation of Cyber Physical Systems (CyPhy) Istanbul, Turkey , July 5-8, 2011
30. Giuseppe Di Guglielmo, Masahiro Fujita, Luigi Di Guglielmo, Franco Fummi, Graziano Pravadelli, Cristina Marconcini and Andreas Foltinek *Model-Driven Design and Validation of Embedded Software* In the proceedings of IEEE/ACM International Workshop on Automation of Software Testing (ICSE Workshop) Waikiki, Honoulu, Hawaii , May 23-24, 2011

31. R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. *The Algorithmic Analysis of Hybrid Systems.* Theoretical Computer Science, 138(1):3–34, 1995.

32. KELLER, R.M. Formal verification of parallel programs. Communications . ACM 19, 7 (July 1976), 371- 384.
33. LAMPORT, L. Proving the correctness of multiprocess programs. IEEE Trans. Software Engineering SE - 3, 2 (Mar. 1977), 125-143.

34. MANSA, Z., AND WALDINGER, R. Is "sometime" sometimes better than "always"? Intermittent assertions in proving program correctness. Commun. ACM 21, 2 (Feb. 1978), 159-172.

35 A. Ferrari, G. Gaviani, G. Gentile, G. Stara, G. Romagnoli, and T. Thomsen. From conception to implementation: a model based design approach. In Proc. of IFAC Symposium on Advances in Automotive Control, 2004.
36. J. R. G. Booch and I. Jacobson. The Unified Modeling Language. Addison-Wesley, 1999.

37. IEC Standard for Property Specification Language (PSL) (Adoption of IEEE Std 1850-2005). IEC 62531:2007 (E), pages 1–156, 2007.
38. Edmund Clarke and Daniel Kroening. Hardware verification using ANSI-C programs as a reference. In Proceedings of ASP-DAC 2003, pages 308–311. IEEE Computer Society Press, January 2003.
39. Simulink® Verification and ValidationTM User's Guide

10 – Appendix

Appx. A

matlab scripts: "mpcBuildSetpoint.m"

```
%==========================================================================
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% START OF MPC PROBLEM SET UP
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% time step
dt = 0.25;
% system matrices
A = [1 dt 0.5*dt*dt;
     0 1 dt;
     0 0 1]; % states [pos; vel; acc]
B = [0.5*dt*dt  0 ;
     dt         0 ;
     1          0 ]; % inputs [delta-acc; con-viol]
C = [0 1 0; % constraints are Cx+Du<=Ymax
     0 -1 0;
     0 0 1;
     0 0 -1;
     0 0 0;
     0 0 0;
     0 0 0];
D = [0 -1 ;
     0 -1 ;
     0 -1 ;
     0 -1 ;
     0 -1 ;
     1  0 ;
    -1  0 ];
Ymax = [0.2; % vel - viol <= 0.2
        0.2;
        0.5; % acc - viol <= 0.1
        0.5;
        0; % viol >= 0;
        0.5 % move <= 0.05
        0.5 ];
% terminal constraints Ex(N)<=XmaxF;
E = 0.00001*[eye(2) [0;0]; -eye(2) [0;0]];
XmaxF = ones(4,1); % effectively relaxed

% cost
% xN*Qf*xN + sum (0.5*x'*Q*x + 0.5*u'*R*u + q'*x + r'*u)
Qf = diag([5 5 2]);
Q = diag([5 5 2]);
R = diag([2 0]);
q = [0; 0; 0];
r = [0; 50]; % large weight on violations
% horizon
N = 10;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% END OF MPC PROBLEM SET UP
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% sizes
Nx = size(A,1);
Nu = size(B,2);
Ny = size(C,1);
% ============ ***Focus on Checkers declaration for code generation***
if size(B,1)~=Nx,
    error('B must have same number of rows as A')
end
if size(A,2)~=Nx,
    error('A must be square')
end
if size(Q,2)~=Nx,
    error('Q must be the same size as A')
end
if size(Q,1)~=Nx,
    error('Q must be the same size as A')
end
if size(C,2)~=Nx,
    error('C must have same number of columns as A')
end
if size(D,1)~=Ny,
    error('D must have same number of rows as C')
end
if size(D,2)~=Nu,
    error('D must have same number of columns as B')
end
% build prediction matrices
% [x(1);..;x(N)] = Phi*x(0) + Gamma*[u(0);..;u(N-1)]
Gamma = B;
Phi = A;
for kk=1:(N-1),
    Phi = [A; Phi*A];
```

```
    Gamma = [Gamma 0*Gamma(:,end-Nu+1:end);A*Gamma(end-Nx+1:end,:) B];
end

% partition for terminal x(N) and intermediate steps
PhiI = Phi(1:end-Nx,:);
GammaI = Gamma(1:end-Nx,:);
PhiF = Phi(end-Nx+1:end,:);
GammaF = Gamma(end-Nx+1:end,:);
% compile QP matrices
% J = 0.5*z'*HH*z + x0*FF*z + gg'*z
% AA*z + CC*x0 <= bb
blk_Q = kron(eye(N-1),Q);
blk_R = kron(eye(N),R);
blk_q = kron(ones(N-1,1),q);
blk_r = kron(ones(N,1),r);
blk_C = kron(eye(N-1),C);
blk_D = kron(eye(N),D);
HH = blk_R + GammaI'*blk_Q*GammaI + GammaF'*Qf*GammaF;
FF = PhiI'*blk_Q*GammaI + PhiF'*Qf*GammaF;
gg = GammaI'*blk_q + blk_r;
% with terminal constraints
AA = [[zeros(Ny,N*Nu);
       blk_C*GammaI] + blk_D;
       E*GammaF];
CC = [C;
       blk_C*PhiI;
       E*PhiF];
bb = [kron(ones(N,1),Ymax);
       XmaxF];
% without terminal constraints
AA = [[zeros(Ny,N*Nu);
       blk_C*GammaI] + blk_D];
CC = [C;
       blk_C*PhiI];
bb = [kron(ones(N,1),Ymax)];
```

==========

matlab scripts: "fixSoftCons.m"

==========

```
function zOut = fixSoftCons(x0,AAs,bbs,CCs,zIn)
%#codegen

% strip slack
zBas = zIn(1:end-1);

% get max constraint violation
viol = max([0; AAs*zBas + CCs*x0 - bbs]);

% reform with increased slack if necessary
zOut = [zBas; viol+1e-5];
```

==========

matlab scripts: "*mpqpSolve.m*"

==========

```
function zOpt = mpqpSolve(x0,HH,FF,gg,AA,bb,CC,zGuess,nIters)
%#eml%
% min 0.5*z'*HH*z + (FF'*x0 + gg)'*z
% s.t. AA*z <= bb-CC*x0
% x0
% initial guess for interior point
z = zGuess;
% iteration counts
nBarrier = nIters(1); %4;
nConjugate = nIters(2); %12;
nBackTrack = nIters(3); %10;
% dummy grad store
lastGrad = gg;
lastStep = gg;
% premultiply to save time
gg0 = FF'*x0 + gg;
bb0 = bb - CC*x0;
% fixed scalar barrier weight
mu = 1;
% choose mu to minimize initial gradient
%mu = 0.001*ones(length(bb),1) - (AA'*diag(1./f))\(HH*z + gg0);
% choose mu to match initial barrier
%mu = f;
% mu to make barrier cost and obj cost same order
f = -1*(AA*z-bb0);
if all(f>0),
    mu = abs((0.5*z'*HH*z + gg0'*z)/(-sum(log(f))));
%else
```

```
    % just show f
    %f
end
% golden ratio
golden = 0.5*(1+sqrt(5));
% barrier solve iteration
for ii=1:nBarrier,
    % conjugate solve iteration
    for jj=1:nConjugate,
        % check feasibility - should be positive
        f = -1*(AA*z-bb0);
        % only try to improve if not infeasible
        if all(f>0),
            % cost
            cost = 0.5*z'*HH*z + gg0'*z - sum(mu.*log(f));
            newCost = cost;
            % gradient
            grad = HH*z + gg0 + AA'*(mu./f);
            % get step direction
            if mod(jj,length(z))==1,
                %disp('Steepest')
                stepDir = -grad;
            else
                % conjugate direction
                beta = (grad'*grad)/(lastGrad'*lastGrad);
                stepDir = -grad + beta*lastStep;
            end
            % store for next time
            lastStep = stepDir;
            lastGrad = grad;
            % overwrite with Newton step dir
            %hess = HH + mu*AA'*diag((1./f))*diag((1./f))*AA
            %stepDir = -hess\grad;
            % initial step - furthest move before hitting constraints
            % i.e. smallest positive value in e
            e = f./(AA*stepDir);
            % st = double(0.999*min(e + 3*max(abs(e))*(e<0)));
            % alternative way of calculating
            st = 1000;
            for ei = e',
                if (ei>0) && (ei<st),
                    st = ei;
                end
            end
            % initial bounds
            ub = 0.999*st; % leave a margin so not right at the constraint
            lb = 0;
            % inner point
            ip = lb + (ub-lb)/golden;
            % only proceed with line search if upper bound still feasible
            if all(bb0-AA*(z+ub*stepDir)>0),
                % costs
                lc = cost;
                uc = 0.5*(z+ub*stepDir)'*HH*(z+ub*stepDir) +
gg0'*(z+ub*stepDir) - sum(mu.*log(bb0-AA*(z+ub*stepDir)));
                ic = 0.5*(z+ip*stepDir)'*HH*(z+ip*stepDir) +
gg0'*(z+ip*stepDir) - sum(mu.*log(bb0-AA*(z+ip*stepDir)));
                for ss=1:nBackTrack,
                    % backtrack if not suitable interval
                    if ic>lc,
                    %if 1<0, % disable backtrack step for testing
                        % cut interval
                        ub = ub*0.2;
                        ip = ip*0.2;
                        uc = 0.5*(z+ub*stepDir)'*HH*(z+ub*stepDir) +
gg0'*(z+ub*stepDir) - sum(mu.*log(bb0-AA*(z+ub*stepDir)));
                        ic = 0.5*(z+ip*stepDir)'*HH*(z+ip*stepDir) +
gg0'*(z+ip*stepDir) - sum(mu.*log(bb0-AA*(z+ip*stepDir)));
                    else
                        % new point
                        np = lb + (ub-ip);
                        nc = 0.5*(z+np*stepDir)'*HH*(z+np*stepDir) +
gg0'*(z+np*stepDir) - sum(mu.*log(bb0-AA*(z+np*stepDir)));
                        % test if new point better or worse than inner point
                        if nc<ic,
                            % new point is better
                            % old inner point becomes new bound
                            if ip>np,
```

```
                                ub = ip;
                                uc = ic;
                            else
                                lb = ip;
                                lc = ic;
                            end
                            % and new point becomes new inner point
                            ip = np;
                            ic = nc;
                        else
                            % new point is worse
                            % new point becomes new bound
                            if np>ip,
                                ub = np;
                                uc = nc;
                            else
                                lb = np;
                                lc = nc;
                            end
                        end
                    end
                end % for ss=1:nBackTrack,
                % check it all worked
                if ic<cost,
                    % update
                    z = z+ip*stepDir;
                %else
                    % just show a message
                    %skip=1
                end
            end % if all(bb0-AA*(z+ub*stepDir)>0),
        end
    end
    mu = 0.33*mu;
end
zOpt = z;
```

Appx. B

```
>> mpcBuildSetpoint
```

Warning: Assertion detected in 'mpcSetpointIntpt/CostOptimizer_propertu/Assertion_OOO_ CostOptimizer' at time 29.5
Warning: Assertion detected in 'mpcSetpointIntpt/CostOptimizer_propertu/Assertion_OOO_ CostOptimizer' at time 33

Warning: Assertion detected in 'mpcSetpointIntpt/CostOptimizer_propertu/Assertion_OOO_ CostOptimizer' at time 36
Warning: Assertion detected in 'mpcSetpointIntpt/CostOptimizer_propertu/Assertion_OOO_ CostOptimizer' at time 36.25
Warning: Assertion detected in 'mpcSetpointIntpt/CostOptimizer_propertu/Assertion_OOO_ CostOptimizer' at time 37.75
Warning: Assertion detected in 'mpcSetpointIntpt/CostOptimizer_propertu/Assertion_OOO_ CostOptimizer' at time 38.5
Warning: Assertion detected in 'mpcSetpointIntpt/CostOptimizer_propertu/Assertion_OOO_ CostOptimizer' at time 39.25

Warning: Assertion detected in 'mpcSetpointIntpt/CostOptimizer_propertu/Assertion_OOO_ CostOptimizer' at time 39.75
Warning: Assertion detected in 'mpcSetpointIntpt/CostOptimizer_propertu/Assertion_OOO_ CostOptimizer' at time 40
Warning: Assertion detected in 'mpcSetpointIntpt/CostOptimizer_propertu/Assertion_OOO_ CostOptimizer' at time 49.25
Warning: Assertion detected in 'mpcSetpointIntpt/CostOptimizer_propertu/Assertion_OOO_ CostOptimizer' at time 53.5
Warning: Assertion detected in 'mpcSetpointIntpt/CostOptimizer_propertu/Assertion_OOO_ CostOptimizer' at time 56

Warning: Assertion detected in
'mpcSetpointIntpt/CostOptimizer_propertu/Assertion_OOO_ CostOptimizer' at time 57.75
Warning: Assertion detected in
'mpcSetpointIntpt/CostOptimizer_propertu/Assertion_OOO_ CostOptimizer' at time 58.25
Warning: Assertion detected in
'mpcSetpointIntpt/CostOptimizer_propertu/Assertion_OOO_ CostOptimizer' at time 58.75
Warning: Assertion detected in
'mpcSetpointIntpt/CostOptimizer_propertu/Assertion_OOO_ CostOptimizer' at time 59.25
Warning: Assertion detected in
'mpcSetpointIntpt/CostOptimizer_propertu/Assertion_OOO_ CostOptimizer' at time 59.75

Warning: Assertion detected in
'mpcSetpointIntpt/CostOptimizer_propertu/Assertion_OOO_ CostOptimizer' at time 60
Warning: Assertion detected in
'mpcSetpointIntpt/CostOptimizer_propertu/Assertion_OOO_ CostOptimizer' at time 78
Warning: Assertion detected in
'mpcSetpointIntpt/CostOptimizer_propertu/Assertion_OOO_ CostOptimizer' at time 78.75
Warning: Assertion detected in
'mpcSetpointIntpt/CostOptimizer_propertu/Assertion_OOO_ CostOptimizer' at time 79
Warning: Assertion detected in
'mpcSetpointIntpt/CostOptimizer_propertu/Assertion_OOO_ CostOptimizer' at time 93

Warning: Assertion detected in
'mpcSetpointIntpt/CostOptimizer_propertu/Assertion_OOO_ CostOptimizer' at time 95.5
Warning: Assertion detected in
'mpcSetpointIntpt/CostOptimizer_propertu/Assertion_OOO_ CostOptimizer' at time 96.75
Warning: Assertion detected in
'mpcSetpointIntpt/CostOptimizer_propertu/Assertion_OOO_ CostOptimizer' at time 115.25
Warning: Assertion detected in
'mpcSetpointIntpt/CostOptimizer_propertu/Assertion_OOO_ CostOptimizer' at time 116.25
Warning: Assertion detected in
'mpcSetpointIntpt/CostOptimizer_propertu/Assertion_OOO_ CostOptimizer' at time 135

Warning: Assertion detected in 'mpcSetpointIntpt/Assertion_VVV_velocity' at time 147.5
Warning: Assertion detected in 'mpcSetpointIntpt/Assertion_VVV_velocity' at time 190.25
==========

Appx. C

==========
>> mpcBuildSetpoint
Starting build procedure for model: mpcSetpointIntpt
Generating code into build folder: C:\Users\marek7\Desktop\mpc_final\mpcSetpointIntpt_grt_rtw
Invoking Target Language Compiler on mpcSetpointIntpt.rtw
Using System Target File: C:\Program Files\MATLAB\R2012a\rtw\c\grt\grt.tlc
Loading TLC function libraries
.....
Initial pass through model to cache user defined code
.....
Caching model source code
.....
Writing source file mpcSetpointIntpt.c
Writing header file mpcSetpointIntpt_private.h
Writing header file mpcSetpointIntpt.h
.
Writing header file mpcSetpointIntpt_types.h

```
### Writing header file CostOptimizer_propertu.h
### Writing header file rtwtypes.h
.
### Writing header file rt_nonfinite.h
### Writing source file rt_nonfinite.c
### Writing header file rt_assert.h
.
### Writing header file rtGetInf.h
### Writing source file rtGetInf.c
### Writing header file rtGetNaN.h
.
### Writing source file rtGetNaN.c
### Writing source file CostOptimizer_propertu.c
### Writing header file rtmodel.h
.
### Writing source file mpcSetpointIntpt_data.c
### Writing header file rt_defines.h
### TLC code generation complete.
.### Creating project marker file: rtw_proj.tmw
### Creating MSVC Solution for model: 'mpcSetpointIntpt'.
### Building Solution in MSVC for model: 'mpcSetpointIntpt'

....
### Saving executable in:
'C:\Users\marek7\Desktop\mpc_final\mpcSetpointIntpt_grt_rtw\msvc\x64\Debug\mpcSetpointIntpt.exe'
### Successful completion of build procedure for model: mpcSetpointIntpt
```

Appx. D

```
#define MODEL mpcSetpointIntpt
#define NUMST 3
#define NCSTATES 2
#define HAVESTDIO
#define RT
#define USE_RTMODEL
#define UNIX //Added in linuxOS for matlab
#define INTEGER_CODE 0
#define MT 1
```

Appx. E

```
                    ====== output of MPC_final===GCOV xxxx.c ======
marek@marekPC15Z:~/Desktop/MPC_final/GCOV_LCOV/all_in1_sin$ gcov CostOptimizer_propertu.c
CostOptimizer_propertu.gcno:version '404*', prefer '406*'
CostOptimizer_propertu.gcda:version '404*', prefer version '406*'
File 'CostOptimizer_propertu.c'
Lines executed:97.06% of 34
CostOptimizer_propertu.c:creating 'CostOptimizer_propertu.c.gcov'

marek@marekPC15Z:~/Desktop/MPC_final/GCOV_LCOV/all_in1_sin$ gcov mpcSetpointIntpt_data.c
mpcSetpointIntpt_data.gcno:cannot open graph file
marek@marekPC15Z:~/Desktop/MPC_final/GCOV_LCOV/all_in1_sin$ gcov rtGetInf.c
rtGetInf.gcno:version '404*', prefer '406*'
rtGetInf.gcda:version '404*', prefer version '406*'
File 'rtGetInf.c'
```

```
Lines executed:78.95% of 38
rtGetInf.c:creating 'rtGetInf.c.gcov'

marek@marekPC15Z:~/Desktop/MPC_final/GCOV_LCOV/all_in1_sin$ gcov rtGetNaN.c
rtGetNaN.gcno:version '404*', prefer '406*'
rtGetNaN.gcda:version '404*', prefer version '406*'
File 'rtGetNaN.c'
Lines executed:80.00% of 25
rtGetNaN.c:creating 'rtGetNaN.c.gcov'

marek@marekPC15Z:~/Desktop/MPC_final/GCOV_LCOV/all_in1_sin$ gcov rt_logging.c
rt_logging.gcno:version '404*', prefer '406*'
rt_logging.gcda:version '404*', prefer version '406*'
File 'rt_logging.c'
Lines executed:22.90% of 1694
rt_logging.c:creating 'rt_logging.c.gcov'

marek@marekPC15Z:~/Desktop/MPC_final/GCOV_LCOV/all_in1_sin$ gcov rt_nonfinite.c
rt_nonfinite.gcno:version '404*', prefer '406*'
rt_nonfinite.gcda:version '404*', prefer version '406*'
File 'rt_nonfinite.c'
Lines executed:75.00% of 16
rt_nonfinite.c:creating 'rt_nonfinite.c.gcov'

marek@marekPC15Z:~/Desktop/MPC_final/GCOV_LCOV/all_in1_sin$ gcov mpcSetpointIntpt.c
mpcSetpointIntpt.gcno:version '404*', prefer '406*'
mpcSetpointIntpt.gcda:version '404*', prefer version '406*'
File 'mpcSetpointIntpt.c'
Lines executed:94.71% of 718
mpcSetpointIntpt.c:creating 'mpcSetpointIntpt.c.gcov'

marek@marekPC15Z:~/Desktop/MPC_final/GCOV_LCOV/all_in1_sin$ gcov grt_main.c
grt_main.gcno:version '404*', prefer '406*'
grt_main.gcda:version '404*', prefer version '406*'
File 'grt_main.c'
Lines executed:48.89% of 135
grt_main.c:creating 'grt_main.c.gcov'

marek@marekPC15Z:~/Desktop/MPC_final/GCOV_LCOV/all_in1_sin$ gcov rt_sim.c
rt_sim.gcno:version '404*', prefer '406*'
rt_sim.gcda:version '404*', prefer version '406*'
File 'rt_sim.c'
Lines executed:80.00% of 5
rt_sim.c:creating 'rt_sim.c.gcov'
```

Appx. F

-fwrapv

This option instructs the compiler to assume that signed arithmetic overflow of addition, subtraction and multiplication wraps around using twos-complement representation. This flag enables some optimizations and disables others. This option is enabled by default for the Java front-end, as required by the Java language specification.

-ansi

This flag tells the compiler to enforce ANSI C standards

-fwrapv

This option instructs the compiler to assume that signed arithmetic overflow of addition, subtraction and multiplication wraps around using twos-complement representation. This flag enables some optimizations and disables others. This option is enabled by default for the Java front-end, as required by the Java language specification.

-fprofile-arcs produces a tally of the execution of each *arc* of the code–one .gcda file for each source file. The --ftest-coverage flag produces .gcno files, which link an *arc* to a source line so that we can see which lines were touched. We'll watch these files being produced in a little bit; you can read about the flags at length in the GNU documentation.

==========

---------- * ----------