

# Static Stability Analysis of Embedded, Autocoded Software

Eric Feron\* and Arnaud Venet†

## 1 Introduction

Embedded software-based control systems are commonly constructed using model-based design environments such as MATLAB/Simulink<sup>TM</sup> from MathWorks. These environments allow the system designer to establish critical properties ensuring the reliability of the system (stability, disturbance rejection, etc.) directly at the model level, using a rich mathematical toolset. However, the software implementation substantially transforms the mathematical model by introducing numerous programming artifacts (aggregate data structures, pointers) and altering the numerical representation (platform-dependent floating/fixed-point arithmetic, and, in the most extreme cases, conversion from continuous-time dynamics to discrete-time dynamics). Verifying that the reliability properties of the system are preserved by the implementation is extremely challenging, yet in many cases critically important. Model-based design environments usually come with an *autocoder* i.e., a code generation tool that automatically synthesizes an implementation of the embedded controller from the specification of its model. Autocoders are getting increasingly used in practical applications for they greatly simplify the implementation process. In many corporations however, including aerospace and automotive industries, autocoding is essentially precluded because its properties are considered to be not adequately trustworthy.

Static program analysis tools have recently proven successful in tackling the certification of embedded software-based control systems. ASTREE [3], developed by P. Cousot's team in France, can automatically verify the consistency of floating-point arithmetic in the electric-command control system of the A380, Airbus' super-jumbo carrier. C Global Surveyor [7], developed by Kestrel Technology LLC, can verify the absence of pointer manipulation errors in the mission-control software of NASA's Mars Exploration Rovers. However, the scope of static analysis has been essentially limited to robustness properties, i.e., ensuring the absence of runtime errors during the execution of the program. Verifying functional properties by static analysis for a system to be used in the field requires (1) translating a reliability property of the model into the implementation setting, using the appropriate data structures and numerical

---

\*Georgia Institute of Technology, [feron@gatech.edu](mailto:feron@gatech.edu)

†Kestrel Technology, 4984 El Camino Real #230, Los Altos, CA 94022, [arnaud@kestreltechnology.com](mailto:arnaud@kestreltechnology.com)

libraries, and (2) tracking the evolution of this property over all execution paths using abstract interpretation techniques. This process requires a tight coupling between the model description and its implementation. While such tight coupling is rarely achieved in practice, it can exist at least when the implementation is automatically generated from the model. Autocoders, like Real-Time Studio<sup>TM</sup> for Simulink<sup>TM</sup>, are increasingly being used in industry for the development of embedded control software. This means that static analysis techniques specialized for codes automatically generated from high-level models can be developed to meet market needs.

The approach we are investigating consists of translating the formal proof of reliability properties of an embedded logic into a dedicated static analyzer that automatically carries out the corresponding proof on the code generated from the model. Whereas today's commercial (and even known academic) static analyzers for embedded mission- and safety-critical software are handcrafted, we propose to use our prior research to construct the dedicated static analyzer automatically. Ultimately, the system designer would be provided with a fully automated engine that performs verification of the generated code without requiring any additional information other than the high-level model specification.

The paper is organized as follows. In Sect. 2 we describe the challenges posed by the analysis of autocoded embedded control software and how they are addressed in our approach. In Sect 3 we propose a research agenda toward the analysis of reliability properties for embedded control software. Section 4 discusses current issues that come up during the design of the static analysis.

## 2 Challenges

There are two major challenges in developing a tool for the automated verification of reliability properties of embedded control software automatically generated from a high-level model:

1. How do we translate a property of the high-level model into a property of the code generated from that model? What are the features of the autocoder we must know to effectively build this translator?
2. How do we specify the basic components of the static analyzer required to verify the desired property? How do we specialize the analyzer to the code generated by a given autocoder?

We discuss these challenges in the following subsections.

### 2.1 Property Translator

The autocoder of the model-based design environment generates code from the model in a predictable way. Therefore, we expect to be able to map a property of the model's variables

into a property of the data structures used in the model’s implementation. Commercial model-based design environments offer rich libraries of basic components for building systems and the properties of interest may greatly vary depending on the nature of the system designed.

The model-based design environment we have chosen is the Matlab/Simulink<sup>TM</sup> tool suite; the family of systems we are beginning with is that of closed-loop dynamical systems, represented on the one hand by a family of differential equations that capture the system’s physics, and on the other hand the closed-loop control algorithm and its associated code. The functional properties we are interested in include closed-loop systems stability, closed-loop system performance (eg tracking performance), and reachability analyses. We are currently interested in describing how to map that property to the implementation by conducting an extensive study of the code generated from a benchmark of systems in that family.

## 2.2 Static Analysis Specification Framework

Checking the transposed property of the model on the code may require a specific analysis algorithm that takes into account the underlying computational model (floating/fixed-point) and the nature of the reliability property (linear or ellipsoidal invariants). Moreover, the code generated by the model-based design environment may use programming language constructs (like pointers or union types in C) that pose a difficulty for the static analyzer. These constructs may require dedicated analysis algorithms (pointer analysis, type analysis) specially tailored for the particular structure of code produced by the autocoder. These static analyzers are strongly specialized toward the family of models and properties considered. This does not require rewriting the analyses from scratch for each of these configurations. We need a library of baseline static analysis algorithms (pointer analysis, floating-point analysis, type analysis, etc.) and a framework for combining them in a way to address each configuration’s unique requirements. We are in the process of establishing a taxonomy of static analysis algorithms and describing a specification framework for expressing arbitrary combinations of these algorithms.

## 3 Research Agenda

### 3.1 Overview of the problem: Illustrative example

Consider a dynamical system described by the following equations of motion

$$\begin{aligned} \frac{d}{dt}x &= y \\ \frac{d}{dt}y &= u + w \end{aligned} \tag{1}$$

where  $x$  is the variable to be controlled,  $u$  is a control variable, and  $w$  a disturbance whose value is unknown (for example wind gusts). We assume a sensor system is able to read  $x$ .

A typical embedded system design problem is to come up with a logic that, based upon successive readings of  $x(t)$  over time, is able to generate a sequence of control inputs  $u(t)$  such that the “closed-loop” system is stable (that is, the variable  $x$  never grows out of bounds), and achieves proper *disturbance rejection*, that is, minimizes the deviations of  $x$  generated by  $w$ . Control systems engineering provides a convenient framework to achieve such performance requirements, by using linear dynamical systems as a modeling framework for such a logic. For example, the logic may be specified by the dynamical system:

$$\begin{aligned} \frac{d}{dt}x_1 &= y, \\ \frac{d}{dt}x_2 &= -10x_2 + y, \\ u &= -0.01x_1 + \frac{9}{10}x_2 - 10y \end{aligned} \tag{2}$$

and it is indeed easy to check (via a simple eigenvalue computation, for example) that the system *specification* consisting of (3) and (2) is stable, and therefore correctly achieves the most important requirement for the embedded system. An alternative and more powerful stability criterion consists of computing an *invariant* for (3) together with (2). Such an invariant might easily be obtained by considering the evolution over time of  $V(x, y, x_1, x_2)$ , with

$$V(x, y, x_1, x_2) = \begin{bmatrix} x \\ y \\ x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 768.5818 & -0.5000 & 0.8254 & -6.3254 \\ -0.5000 & 82.5378 & 50.0000 & 6.7959 \\ 0.8254 & 50.0000 & 495.5018 & 4.4932 \\ -6.3254 & 6.7959 & 4.4932 & 0.6616 \end{bmatrix} \begin{bmatrix} x \\ y \\ x_1 \\ x_2 \end{bmatrix}$$

The level sets of  $V$  are ellipsoids, and, in the absence of perturbations  $w$ , the system (3) together with the logic specification (2) satisfies

$$\frac{d}{dt}V \leq 0$$

for any initial condition. This in turn ensures all variables of the closed-loop system specification remain bounded, and therefore the system is stable.

Consider now the *implementation* of the logic (2): This implementation must replace the continuous-time system (3) by a difference scheme instead, with or without variable time step. Other significant differences may exist, including the coding of all variables in floating-point arithmetic. The invariant function  $V$  may, however, still be used to prove the stability of the system (3) together with the implementation of the controller specification (2). This is what we propose to achieve by conducting static analysis on the code generated from the model, using Abstract Interpretation techniques. Kestrel Technology has a substantial body of implemented analyzers based on abstract interpretation and libraries [7, 6] available. These algorithms are being incorporated in *CodeHawk*, a generic static analysis development framework that enables the automated construction of static analyzers from a set of requirements.

### 3.2 From system-level properties to implementation-level properties

In the case of the reliability property presented in the previous section (stability), we have to prove the preservation of an ellipsoidal invariant. This property is expressed on a continuous

model and has to be transformed so that it can be checked directly on the implementation of a discrete model extracted from the continuous formulation. This process can be split up in two separate steps:

- (1) Generate a discrete executable model from a continuous formulation.
- (2) Translate the discrete executable model into an actual program.

The first step is a standard mathematical transformation that is fairly independent from the choice of a particular autocoder. In order to assess the feasibility of our approach, we are designing a static analysis framework that operates on the executable model of (1). This executable model can be seen as the most detailed operational formulation that can be stated independently from the characteristics of the target programming language. Since this formulation does not use low-level programming language constructs we can focus on the algorithmic aspects of the static analysis. We assign a syntax and semantics to this operational model in order to conduct formal reasoning on it and define its abstract interpretation rigorously. The connection between this intermediate operational model and the actual implementation is investigated independently by studying the code generated from a benchmark of representative models that we are building build for that purpose.

### 3.3 Tailoring the analyzer to the property to verify

The executable model of the system introduces a complex control-flow with loops that represents the iterative computation of the control logic over discretized time. This has the effect of breaking down the simple formulation of the original logic into a number of intermediate steps. This means that the original invariant that holds at some points in the executable code does not hold at some others. Therefore, the analysis must be able to model how the invariant is transformed by elementary operations of the executable model. We need what is called in the jargon of Abstract Interpretation an *abstract domain*. In the case of ellipsoidal invariants, we need to construct an abstract domain that can represent all possible ellipsoidal invariants over a given set of variables as well as the associated semantic transformers. The semantic transformers model the effect of elementary operations like initializing a variable or incrementing its value.

The executable model on which we are working hides low-level implementation details, there is one feature of the target platform that we must take into account at this level because of its impact on the design of the abstract domain: the computational model of reals (floating/fixed-point arithmetic). How roundoffs are performed and imprecision is propagated is extremely important, because the accumulation of roundoff errors can cause a violation of the invariant and compromise the stability of the control system. This issue extends to other discretizations that may be performed automatically, such as time discretization. Tracking the propagation of roundoff and discretization errors requires a proper abstraction of the floating-point computational model that is amenable to mechanized analysis.

The design of the abstract domain is the most critical aspect of our approach. It requires expertise both in Abstract Interpretation and Control Theory in order to devise a representation

of invariants that will effectively enable checking the correctness of the executable model with good performance. Fortunately, there is a substantial body of work already published that addresses many the problems involved in the design of such abstract domains, like the inference of numerical invariants for floating-point computations [5] or the discovery of ellipsoidal invariants for the analysis of linear digital filters [4], with extensions to uncertain dynamical systems [1]. Abstract domains specialized for certain properties can be combined in various ways in order to obtain more expressive ones that can handle more complex properties. Therefore, the effort of building a new abstract domain is incremental. We plan to use *CodeHawk* as a repository of abstract domains as well as a generator for building new abstract domains from combination of existing ones.

### 3.4 Analyzing implementation artifacts

Studying the analysis of the executable model provides us with the core concepts for carrying out the verification at the implementation level. In order to translate these concepts into an actual analyzer that operates at the generated code level, we need a number of auxiliary analyses whose purpose is to recover structural data from the implementation artifacts. For example, consider the following function which has been taken out of the C code generated by Real-Time Studio from the sample continuous model described in Section 3.1.

```
static void rt_ertODEUpdateContinuousStates(RTWSolverInfo *si , int_T tid)
{
    time_T tnew = rtsiGetSolverStopTime(si);
    time_T h = rtsiGetStepSize(si);
    real_T *x = rtsiGetContStates(si);
    ODE1_IntgData *id = rtsiGetSolverData(si);
    real_T *f0 = id->f[0];
    int_T i;

    int_T nXc = 2;

    rtsiSetSimTimeStep(si,MINOR_TIME_STEP);

    rtsiSetdX(si, f0);
    logic_derivatives();
    rtsiSetT(si, tnew);

    for (i = 0; i < nXc; i++) {
        *x += h * f0[i];
        x++;
    }

    rtsiSetSimTimeStep(si,MAJOR_TIME_STEP);
}
```

This function updates the control variables at each time step accordingly to the specified logic. In order to recover the original components of the continuous model we must perform a pointer analysis that is able to distinguish between elements of arrays accessed through pointers (like `x` and `f0`) and a numerical analysis that is able to infer the range of index variables (like `i`) used for manipulating the model data. If we consider pointer analysis for example, there is a broad spectrum of existing algorithms that greatly differ in terms of precision and scalability. Depending on the structure of the code produced by the generator or just the family of models considered, we may have to choose different algorithms. In order to achieve this we need a high level of flexibility for specifying the set of analyses to be performed on the code.

We plan to use the *CodeHawk* static analysis generator that precisely enables the rapid development of efficient analyzers from a list of requirements. What has to be studied, however, is how to state these requirements for this specific application, i.e., how to recover the structure of the high-level model. This means that we have to determine a formalism that both specifies which combination of analysis to use and how to tie the results of the analysis back to the model components. This formalism would ultimately be integrated within *CodeHawk*. We are now studying the features that such a formalism should possess and we will propose a tentative definition from experiments conducted on a benchmark of models using *CodeHawk*.

## 4 Some Current Issues: Designing the Collecting Semantics

When analyzing code, a central issue is the design of what is called the *collecting semantics* [2]. The collecting semantics describes how much information from the original program must be retained in order to verify the desired property. The collecting semantics forms the base model on which static analysis is conducted. Higher levels of semantic collection allow one to define more compact models of the software execution, but this task may also be more complex, since information must be collected over several lines of code and then linked into a compact model. Thus, lower semantic levels (like “line-by-line” analyses) are more desirable from the standpoint of analyzer simplicity and adaptability. We now show on a simple example that testing typical control-related invariants raises interesting issues with respect to the collecting semantics.

Consider a C code implementation of the simple recursion

$$\begin{aligned} &\text{for(;;) } \{ \\ &\quad x_+ = Ax; \\ &\quad x = x_+; \\ &\} \end{aligned} \tag{3}$$

In this infinite recursion,  $x \in \mathbf{R}^n$  is the state of the system, and the matrix  $A \in \mathbf{R}^{n \times n}$  is a square matrix. We assume the matrix  $A$  to be stable, that is, all trajectories  $x$  converge to zero, which is equivalent to the spectral radius of  $A$  being strictly less than 1. A typical invariant used to prove this property is a quadratic function of the state  $x$ , that is, a function of the form  $V(x) = x^T P x$ , where  $P \in \mathbf{R}^{n \times n}$  is a positive definite, symmetric matrix. Such quadratic

invariants can always be found if the matrix  $A$  is stable, and the monotonic decay of  $V$  is equivalent to the matrix inequality (in the sense of the partial order of symmetric matrices):

$$A^T P A - P \leq 0$$

Assume such a matrix  $P$  is known. A software implementation of (3) might include several lines of code to carry the line  $x_+ = Ax$ , and then several other lines of code to implement  $x = x_+$ . When  $n > 1$ , the latter command can raise significant issues, especially if they are distributed throughout the code as is often done. Indeed, each line of code corresponds to the substitution of *one entry of the state vector  $x$  at a time*. The effect on the invariant is, most often, disastrous. Indeed, considering for example a substitution of pointer values of the kind

```
for (i =0; i < n; i++) {
    *(x + i) = *(x_+ + i);
}
```

Then the value of the invariant  $V$  after the first line of this recursion is

$$\begin{bmatrix} a_1 x \\ x_2 \\ \vdots \\ x_n \end{bmatrix}^T P \begin{bmatrix} a_1 x \\ x_2 \\ \vdots \\ x_n \end{bmatrix},$$

where  $a_1$  is the first row of the matrix  $A$ , and the difference between this and  $V(x) = x^T P x$  is

$$\begin{bmatrix} a_1 x \\ x_2 \\ \vdots \\ x_n \end{bmatrix}^T P \begin{bmatrix} a_1 x \\ x_2 \\ \vdots \\ x_n \end{bmatrix} - x^T P x,$$

or

$$\begin{bmatrix} a_1 x - x_1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}^T P x + x^T P \begin{bmatrix} a_1 x - x_1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} + \begin{bmatrix} a_1 x - x_1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}^T P \begin{bmatrix} a_1 x - x_1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

the last of the three terms is positive, such that the candidate invariant  $x^T P x$  decays if and only if, writing  $a_1 = [a_{11} \ a_{12} \ \dots \ a_{1n}]$  and introducing  $b = [a_{11} - 1 \ a_{12} \ \dots \ a_{1n}]$ , we have, for any  $x$ :

$$x^T b^T p_1 x + x^T p_1^T b x \leq 0$$

where  $p_1$  is the first row of  $P$ . However, it is well known from linear algebra that such an inequality is possible only if  $b$  and  $p_1$  are collinear and oriented in the same direction, which is usually not the case. Thus, the invariant  $x^T P x$  does not decay line-by-line, and we must first collect the *entire* substitution of  $x$  by  $x_+$  before the invariant decays. This substantially constrains the design of the collecting semantics and the static analysis, since all possible execution traces of a loop must be collected and represented by one semantic object.

## Conclusion

Static analysis of embedded software arising from aut coded specifications is a necessary step towards broad and safe acceptance by the industrial community. This paper has outlined a research program aimed at achieving this task by exploiting the underlying structure arising from this type of software, using abstract interpretation and control systems analysis methods. The execution of this research is an ongoing activity and its findings will be reported in future publications.

## References

- [1] BOYD, S., EL GHAOUI, L., FERON, E., AND BALAKRISHNAN, V. Linear matrix inequalities in system and control theory. *SIAM Studies in Applied Mathematics* 15 (1994).
- [2] COUSOT, P. Semantic foundations of program analysis. In *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones, Eds. Prentice-Hall, Inc., Englewood Cliffs, 1981, ch. 10, pp. 303–342.
- [3] COUSOT, P., COUSOT, R., FERET, J., MAUBORGNE, L., MINÉ, A., MONNIAUX, D., AND RIVAL, X. The ASTRÉE Analyser. In *Proceedings of the European Symposium on Programming (ESOP'05)* (2005), vol. 3444 of *Lecture Notes in Computer Science*, pp. 21–30.
- [4] FERET, J. Static analysis of digital filters. In *European Symposium on Programming (ESOP'04)* (2004), no. 2986 in LNCS, Springer-Verlag.
- [5] MINÉ, A. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP'04* (2004), vol. 2986 of LNCS, Springer, pp. 3–17.
- [6] VENET, A. A scalable nonuniform pointer analysis for embedded programs. In *Proceedings of the International Static Analysis Symposium, SAS 04* (2004), vol. 3148 of *Lecture Notes in Computer Science*, Springer, pp. 149–164.
- [7] VENET, A., AND BRAT, G. Precise and efficient static array bound checking for large embedded C programs. In *Proceedings of the International Conference on Programming Language Design and Implementation* (2004), pp. 231–242.