2014 American Control Conference (ACC)
June 4-6, 2014. Portland, Oregon, USA

Credible Autocoding of Fault Detection Observers

Timothy E. Wang[1], Alireza Esna Ashari[1], Romain J. Jobredeaux[1], and Eric M. Feron[1]

Abstract— **We present a domain specific process to enable the verification of observer-based fault detection software. Observer-based fault detection systems, like control systems, yield invariant properties of quadratic types. These quadratic invariants express both safety properties of the software, such as the boundedness of the states, and correctness properties, such as the absence of false alarms from the fault detector. We seek to leverage these quadratic invariants, in an automated way, for the formal verification of the fault detection software. The approach, named the credible autocoding framework, can be characterized as autocoding with proofs. The process starts with the fault detector model, along with its safety and correctness properties, all expressed formally in a synchronous modeling environment such as Simulink. The model is then transformed by a prototype credible autocoder into both code and analyzable annotations for the code. We demonstrate the credible autocoding process on a running example of an output observer fault detector for a 3 degree-of-freedom helicopter control system.**

Keywords: Fault Detection, Software Verification, Credible Autocoding, Aerospace Systems, Formal Methods.

I. Introduction

Safety-critical systems of a cyber-physical nature are increasingly present in a number of industries. Whether it is the flight control system of an aircraft, or the digital controller of a pacemaker, a failure in such systems can have significant costs, both monetary and in terms of human lives. For this reason, the field of fault detection has naturally turned its attention to these systems [1], [2], [3], [4], [5]. Most of the current body of work develops observer-based fault detection methods, which are suitable for online fault detection in the case of abrupt faults. The observer provides analytic redundancy for the dynamics of the system. Comparing the input-output data with the nominal data obtained from the model of the system, we conclude whether or not the system is in nominal mode. Possible faults are modeled as additive inputs to the system. Such an additive fault changes the nominal relations between inputs and outputs. An overview of the recent developments in this domain can be found in [6], [7], [8]. Observer-based fault detection methods are usually implemented as software on digital computers. However, there is a semantic gap between fault detection theory and the software implementation of these methods. Computation errors may cause incorrect results, and engineers with little or no background in control may need to test and modify the software, hence the need to express fault detection semantics at the level of the code. Additionally, such an endeavor can help verify systematically that the software behaves correctly based on theory and initial design.

In this paper, we present an automated process where control-theoretic techniques are applied towards the verification of observer-based fault detection software. We extend our previous works [9], [10], [11], [12] on controller systems to fault detection systems. The application of system and control theory to control software verification can be traced back to [13], [14]. In these papers, the authors presented an example where a controller program was documented with a quadratic invariant set derived from a stability analysis of the state-space representation of the controller. Since then, progress has been made towards the creation of an automated framework that can rapidly obtain and transform high-level functional properties of the control system into logic statements that are embedded into the generated code in the form of comments. The usefulness of these comments comes from their potential usage in the automatic verification of the code. We will refer to the logic statements as "annotations" and the generated code with those comments as "annotated code". We named the framework *credible autocoding* [12], as it is a process to rapidly generate the software as well as the annotations that guarantee some functional properties of the system. The realization of the framework is a prototype tool that we have built and applied to control systems such as a controller for the 3 degree of freedom Quanser helicopter [15]. For this paper, we have further refined the prototype to handle the addition of a fault detection system running along with the Quanser controller.

II. Credible Autocoding

Credible autocoding is an automatic or semi-automatic process that transforms a system that is initially expressed in a language of high-level of abstraction, along with the mathematical proofs of its good behavior, into code, annotated with said mathematical proof. The initial level of abstraction is the differential equation of the system and the final level could be the software binary. For the prototype implementation of our framework, we chose Simulink as the input language and C code as the output language. Regardless of the input and output languages, the main contribution from this prototype is the automatic translation of fault-detection properties into axiomatic semantics for the output code.

Axiomatic semantics are an approach to reason about the correctness of program that trace back to the work of

*This article was prepared under support from NSF Grant CNS - 1135955 "CPS: Medium: Collaborative Research: Credible Autocoding and Verification of Embedded Software (CrAVES)", NASA Grant NNX12AM52A "Validation Elements For Loss-of-Control Recovery Operations (VELCRO)", the Army Research Office under MURI Award W911NF-11-1-0046, and ANR ASTRID project VORACE.

[1] Department of Aerospace Engineering, Georgia Institute of Technology, Atlanta, GA 30332, USA {timothy.wang,esna_ashari, jobredeaux,feron}@gatech.edu

```
/*
  @ requires x<=0;
  @ ensures x>=0;
*/
{
          x=x*x;
}
```

Fig. 1. C code with ACSL

```
/*
  @ assumes input*input<1;
  @ loop invariant x*x<=1;
*/
{
          while (1) {
                x=0.98*x+0.02*input;
          }
}
```

Fig. 2. C code with ACSL

Hoare [16]. In this approach, the semantics or mathematical meaning of a piece of code are defined through how the piece of code modifies certain logic predicates on the variable(s) of the code.

The basics of axiomatic semantics are demonstrated here using two examples. The piece of C code in figure II computes the square of x and assigns the answer to the variable x. Two logic predicates, $x <= 0$ and $x >= 0$, preceded by the symbols "@ requires" and "@ ensures" appear in the comments or annotations that precede the code. They represent properties of x that we claim to be true, respectively before and after the execution of the line of code. The keyword *requires* denotes a pre-condition and the keyword *ensures* keyword denotes a post-condition. The pre and post-conditions, together with the statement they surround, form a Hoare triple, which is a contract expressing that during any execution of the program, if the pre-condition is true before the statement is executed, then the post-condition will be true after its execution. In this example, it is trivial to see that if the variable x is non-positive before the execution of $x := x * x$ then it will be non-negative afterwards. However we stress here that any such Hoare triples inserted as annotations in the code needs to be formally proven before it is said to be valid.

Consider the C implementation of a 1-dimensional linear state-space system in figure II. The state-transition matrix A is 0.98 and the input matrix B is 0.02. Unlike the previous example, this piece of code contains a loop. When annotating a loop with a Hoare triple, properties of interest need to hold before, throughout and after the execution of the loop. This type of properties are referred to as *inductive invariants*. They represent both a pre and post condition for the body of the loop.

Often, verifying the correctness of even trivial inductive invariants can be a non-trivial task. However, comparatively, it is a much more tractable task to automatically verify the correctness of relevant inductive invariants than to discover them in the first place. Domain specific knowledge is key to finding these invariants. Indeed, for even simple examples, it can be impossible for general automatic decision procedures to compute them. In addition to inductive invariants, we can also express assumptions, such as $input * input < 1$, using the keyword *assumes*, Unlike the inductive invariant $x*x <= 1$, the *assertion* $input * input < 1$ cannot be checked for its correctness using solely the code.

For this example or any other linear state-space systems, we can apply domain specific knowledge, namely, Lyapunov-based theories, to compute an ellipsoid invariant set $\mathcal{E}(x, P) = \{x | x^{\mathrm{T}} P x \leq 1\}$. A collection of this type of quadratic stability results with an efficient computational solutions can be found in [17]. Here, $\mathcal{E}(x, 1)$ forms a valid invariant set. Using the credible autocoding framework, this invariant property can be rapidly transformed into Hoare triples for the code, and thus, in theory, makes the process of automatic verification of the generated code more feasible.

The code annotations in the two examples are expressed in the ANSI C Specification Language (ACSL)[1]. In the latter sections, the autocoded fault detection semantics are also expressed in ACSL. For more details, see [18].

III. Fault detection problem formulation

We focus on observer-based fault detection of dynamic systems. Such methods need the system to be modeled by differential equations. We designed the fault detection observer for a three-degree-of-freedom laboratory helicopter. The system is modeled by nonlinear equations. Such a model can be linearized around the operating point of the system as follows

$$\dot{x}(t) = Ax(t) + Bu(t) + Ef(t), \quad (1)$$

$$y(t) = Cx(t), \quad (2)$$

where $x(t) \in \mathbb{R}^6$ and $u(k) \in \mathbb{R}^2$ are the state vector and the known input vector at time t, respectively. Also, $y(t) \in \mathbb{R}^3$ is the output vector. A, B and C are state transition, input and output matrices, as in [15]. $f(t) \in \mathbb{R}^{n_f}$ in equation (1) represents an additive fault to the system that should be detected. No prior knowledge on this input signal is available. The value of $f(t)$ is zero for nominal (fault-free) system. The aim of the fault detection is to raise an alarm whenever this value differs significantly from zero (faulty system).

We consider an actuator degradation fault for this system. Such a fault changes the behavior and the steady state of the system, and can be modeled as an additive fault. The effect of the degradation can be modeled by replacing $u(t)$ in equation (1) with $\bar{u}(t)$, where

$$\bar{u}(t) = Xu(t). \quad (3)$$

Thus, we obtain the fault matrix below, defined in equation (1)

$$E = B(I - X). \quad (4)$$

[1]The prototype credible autocoder also produces annotations in ACSL

A. Output observer design for fault detection

To describe the autocoding process, we select the simplest observer-based method [6], [8]. The detector observes the system, receives input and output data and compares it with the nominal response of the system.

Consider the full-order state observer below

$$\dot{\hat{x}}(t) = A\hat{x}(t) + Bu(t) + L(y(t) - C\hat{x}(t)), \quad (5)$$
$$\hat{y}(t) = C\hat{x}(t). \quad (6)$$

Using this observer, we generate a residual signal, comparing the estimated output in equation (6) with the measured one

$$r(t) = y(t) - \hat{y}(t). \quad (7)$$

We compare the residual signal $r(t)$ against a predefined threshold. If the threshold is reached, a fault alarm is raised. In order to explain how the method works and how the observer should be designed, we introduce the estimation error $e(t) = x(t) - \hat{x}(t)$ and compute the error dynamics

$$\dot{e}(t) = (A - LC)e(t) + Ef(t), \quad (8)$$
$$r(t) = Ce(t). \quad (9)$$

From equations (8)–(9), $r(t)$ goes to zero if $f(t)$ is zero and the observer matrix L is chosen so that $A - LC$ is stable. Note that L is the only design parameter for this observer. In practice, no fault alarm is raised if $f(t)$ is too small. Thus, we suppose $\|f(t)\| > \sigma$ is a fault that must be detected. Consequently $\|r(t)\| > r_{th}$ raises a fault alarm, where r_{th} is the threshold corresponding to σ.

IV. A formal method to verify fault detection

The theory behind fault detection methods is presented in Section III-A. However, there always exists a semantic gap between theory and real implementation. The methods in Section III-A should be implemented in the form of software, either in a graphical environment such as Simulink or as computer code in a programming language such as Matlab or C. Due to computation errors and the use of floating point numbers in digital computers, there exists a difference between the implemented method and the ideal results of theory. We aim at annotating the software so that an expert or a machine can track its operation and verify that the design criteria are satisfied at the level of the code. The idea developed in [19] to formally document the stability of closed-loop systems is now extended to fault detection methods.

In order to certify the fault detection software, we need to certify particular properties of the observer:

1) Stability: the error dynamics are stable, i.e. $e(t)$ in equation (8) is around the origin when the system is in the nominal case and remains bounded in the faulty case.
2) Fault detection: the residual $r(t)$ correctly detects the fault. In other words $r(t)$ does not reach a predefined threshold if $f(t)$ is sufficiently small.

To verify these properties, we use Lyapunov theory, which was shown to be a good mechanism to generate easy-to-use, formal code annotations (see [19]). Note that Lyapunov functions are used here as opposed to a more classic approach based on the location of the observer poles. This is because the former yield invariant sets that translate well into loop invariants at the code level. This makes it easier to bridge the gap between control theory and computer science. Frequency-domain based analyses on the oher hand, and how they relate to the stability of the system, are not so obviously connected to invariants pertaining to the program variables. We start from the informal specifications in Section III-A and translate them into formal specifications as follows. These properties show how the software variables can change so that the pre-defined filter specifications are verified.

Suppose the system is in nominal mode. Considering a Lyapunov function $V(t) = e^T(t)Pe(t)$, where P is a positive definite matrix. We can show that the $e(t)$ remains in a predefined invariant ellipsoid

$$\mathcal{E}_n = \{e(t) \in \mathbb{R}^n | e^T(t)Pe(t) \leq \zeta\}, \quad (10)$$

for all $t \in \mathbb{R}$ if the observer is stable. Here, $\zeta \geq 0$ is a scalar.

For the faulty mode we can introduce a similar ellipsoid around the new equilibrium point. However, we do not know the new equilibrium point, as the fault is supposed to be completely unknown. But in practice, $f(t)$ is bounded. Suppose that $\|f(t)\| < \sigma$. We introduce

$$\mathcal{E}_f = \{e(t) \in \mathbb{R}^n | e^T(t)Pe(t) \leq \bar{\zeta}\}. \quad (11)$$

In equation (11) $\bar{\zeta}$ is

$$\begin{aligned} \bar{\zeta}(t) = & \max_e e^T(t)Pe(t) \\ & s.t. \quad \dot{e}(t) = (A - LC)e(t) + Ef(t) \\ & and \quad \|f(t)\| < \sigma. \end{aligned} \quad (12)$$

Thus, we have two ellipsoids to which the value of the Lyapunov function may belong. As far as $\forall t$, $V(t) \in \mathcal{E}_n$, the system is in nominal mode and the observer is stable. On the other hand if $\forall t$, $V(t) \in \mathcal{E}_f$, the system is in faulty mode and the observer is stable. If $\exists t, V(t) \notin \mathcal{E}_f$ the detector is unstable.

V. Autocoding of Fault Detection Semantics

In this section, we describe the autocoding of the fault detection semantics of a running example. The running example is a fault detection system as specified in section III-A combined with a LQR controller that has two integrators. We first point out that on the abstraction level of a computer program, the notion of a continuous-time differential equation like in (6) no longer applies. The running example, including the plant, needs to be in discrete-time. In fact, for analysis purposes, the plant can be treated as another C program. We have the following discrete-time linear state-space systems

$$\begin{aligned} x_c(k+1) &= A_c x_c(k) + B_c y(k), \\ u(k) &= C_c x_k + D_c y(k), \end{aligned} \quad (13)$$

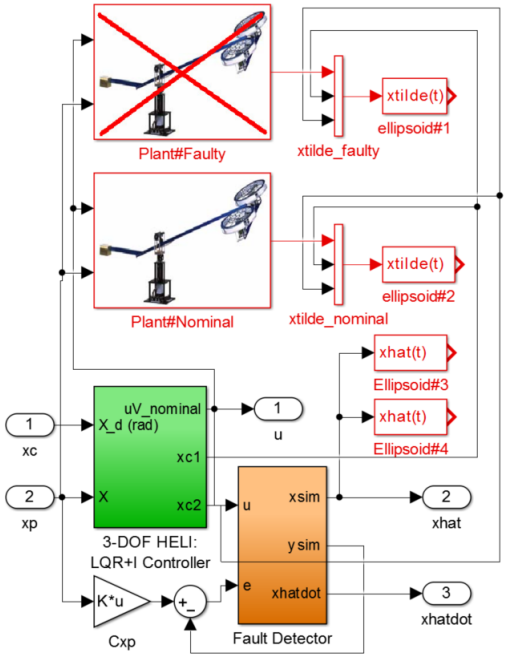

Fig. 3. Simulink Model Input For Credible Autocoding

$$\begin{aligned} \hat{x}(k+1) &= \hat{A}\hat{x}(k) + Bu + Ly, \\ r(k) &= y(k) - \hat{y}(k), \end{aligned} \tag{14}$$

and

$$\begin{aligned} x(k+1) &= Ax(k) + Bu(k), \\ y(k) &= Cx(k), \end{aligned} \tag{15}$$

representing the controller, the detector, and the plant respectively. After discretization, system matrices change. However, with an abuse of notation, we use the same symbols for the discrete-time model of the system and the observer in equations (14)–(15). We also have a discretized version of the error dynamics from equation (6),

$$e(k+1) = \hat{A}e(k) + Ef(k) \tag{16}$$

where $f(k)$ represent sampled fault signal and $\hat{A} = A - LC$.

The Simulink model of the controller and the fault detection system is displayed in Figure 3. The inserted fault detection semantics are expressed using the annotation blocks, which are in red in Figure 3. The annotation blocks are converted into ACSL annotations for the output code. The four Ellipsoid blocks represent four ellipsoid invariants. Two are for the closed loop plant and controller state $\tilde{x}$ and two other for the detector states $\hat{x}$. The semantics of the plants (faulty and nominal) are expressed using the Plant annotation blocks in Figure 3. We do not express the ellipsoid sets for the error states from equations (10) and (11) on the input Simulink model. The reason for this is explained in section (V-A). However, we point out that the credible autocoding process will generate two ellipsoid sets on the error states $e = x - \hat{x}$, one for the nominal plant and the other for the faulty. They are just expressed in the annotations of the generated code output. Not shown in Figure 3 is the ellipsoid observer block expressing a bound on the input signal y_c. This bound is an assumption that gets transformed into an ACSL assertion.

A. *Ellipsoid sets on the error states e*

In this section, we discuss the choice of ellipsoid sets that are annotated on the Simulink model. Instead of using the ellipsoids from equations (10) and (11), which are on the error e, we use two different sets of ellipsoids. One is on the observer states and the other is on the controller states. From those two sets of ellipsoid invariants, the autocoder can generate the desired ellipsoid invariants on the error. The reason for this selection is mainly due to the current technical limitation of the autocoder prototype as the error e does not explicitly appear in either the controller, the observer or the plant. Consider the observer dynamic

$$\hat{x}(k+1) = \hat{A}\hat{x}(k) + Bu(k) + LCx(k). \tag{17}$$

Note that equations (17), and (14) are equivalent. During the credible autocoding process, any ellipsoid invariant that is inserted onto the input model is propagated forward through the generated C code using ellipsoid calculus [20]. Methods such as computing the affine transformation of an ellipsoid are used often, since semantically speaking, most of the generated C code is consisted of assigning affine expressions to variables. For example, if an ellipsoid set $\mathcal{E}(x, P)$ is the pre-condition, and the ensuing block of code is semantically $x := Ax$, then the autocoder generates the post-condition $\mathcal{E}\left(x, \left(AP^{-1}A^{\mathrm{T}}\right)^{-1}\right)$. On the actual C code, the propagation steps are much smaller so there are a sequence of intermediate ellipsoid invariants between $\mathcal{E}(x, P)$ and $\mathcal{E}\left(x, \left(AP^{-1}A^{\mathrm{T}}\right)^{-1}\right)$. Let $\tilde{x} = \begin{bmatrix} x \\ x_c \end{bmatrix}$ be the closed-loop system states, and assume that closed-loop stability analysis yields an ellipsoid invariant set $\mathcal{E}(\tilde{x}, P_0)$. Given $\mathcal{E}(\tilde{x}, P_{\tilde{x}})$, the prototype autocoder can generate P_u, P_x such that $\mathcal{E}(u, P_u)$ and $\mathcal{E}(x, P_x)$. Given $\mathcal{E}(u, P_u)$, $\mathcal{E}(x, P_x)$, and the dynamics in equation (17), one can compute an ellipsoid invariant $\mathcal{E}(\hat{x}, P_{\hat{x}})$ for the detector states $\hat{x}$ by solving a linear matrix inequality (LMI). Solving the LMI also yields the relaxation multipliers $\alpha > 0$ and $\gamma > 0$ for the quadratic inequalities in $\mathcal{E}(u, P_u)$ and $\mathcal{E}(x, P_x)$. These multipliers are used to generate an ellipsoid invariant on e in the following way. Given the ellipsoid invariants $\mathcal{E}(\hat{x}, P_{\hat{x}})$, $\mathcal{E}(x, P_x)$, and the error states $e = \begin{bmatrix} I & -I \end{bmatrix} \begin{bmatrix} x \\ \hat{x} \end{bmatrix}$, a correct ellipsoid invariant on e is $\mathcal{E}(e, P_e)$, where

$$P_e = \left(\begin{bmatrix} I & -I \end{bmatrix} P_{x,\hat{x}}^{-1} \begin{bmatrix} I & -I \end{bmatrix}^{\mathrm{T}} \right)^{-1} \tag{18}$$

with $P_{x,\hat{x}} = \begin{bmatrix} \gamma P_x & 0 \\ 0 & (1-\alpha-\gamma) P_{\hat{x}} \end{bmatrix}$. Hence given the two invariants $\mathcal{E}(\hat{x}, P_{\hat{x}})$, $\mathcal{E}(\tilde{x}, P_{\tilde{x}})$, which are explicitly inserted into the Simulink model, and the prototype autocoder can automatically produce the ellipsoid invariants on the error e.

The annotations generated by the credible autocoding process can guarantee both the safety property (the ellipsoid

bounds on the variables correspond to x_c and $\hat{x}$) and the liveness property of the fault detection system i.e. the two ellipsoids on the error states.

B. Ellipsoid sets in the Simulink model

To generate the ellipsoid invariants for the credible autocoding, we have

$$\hat{x}(k+1) = \hat{A}\hat{x}(k) + \hat{B}\hat{u}(k) \tag{19}$$

with $\tilde{B} = LC$, $\hat{A} = A - LC$, $\hat{u} = \begin{bmatrix} u \\ x \end{bmatrix}$, and $\hat{B} = \begin{bmatrix} B & \tilde{B} \end{bmatrix}$. Given that the closed-loop ellipsoid set $\mathcal{E}(\tilde{x}, P_{\tilde{x}})$ implies $\mathcal{E}(u, P_u)$ and $\mathcal{E}(x, P_x)$ for some matrices P_u, P_x by the affine transformation of ellipsoid set. With $\mathcal{E}(u, P_u)$, $\mathcal{E}(x, P_x)$, and the detector dynamics in equation (17), we have the following results for computing an ellipsoid invariant on $\hat{x}$.

Lemma V.1 *Let* $\hat{u} = \begin{bmatrix} u \\ x \end{bmatrix}$ *and assume that* $\hat{u}$ *belongs to the set* $\{\hat{u} | \hat{u}^T P_1 \hat{u} \leq 1\}$. *If there exist a symmetric positive-definite matrix* P *and a positive scalar* α *that satisfies the following linear matrix inequality*

$$\begin{bmatrix} \hat{A}^T P \hat{A} - P + \alpha P & \hat{A}^T P \hat{B} \\ \hat{B}^T P \hat{A} & \hat{B}^T P \hat{B} - \alpha P_1 \end{bmatrix} \prec 0 \tag{20}$$

then the set $\{\hat{x} | \hat{x}^T P \hat{x} \leq 1\}$ *is invariant with respect to equation (19).*

First we manually compute the invariant sets for the closed-loop system. Once for the faulty plant and once more for the nominal plant.For the closed-loop analysis, we assume the command input y_c is bounded. From the obtained closed-loop invariant sets, the autocoder can generate two ellipsoid invariants on $\hat{u}$, $\mathcal{E}(\hat{u}_i, P_{\hat{u}_i}, i = N, F)$, where N, F denote respectively nominal or faulty. Now we apply lemma V.1 twice to obtain the two ellipsoid sets $\mathcal{E}(\hat{x}_i, P_{\hat{x}_i}), i = N, F$ on $\hat{x}$. As discussed before, we insert the obtained ellipsoid invariants $\mathcal{E}(\tilde{x}_n, P_{\tilde{x}_i}), i = N, F$ and $\mathcal{E}(\hat{x}_i, P_{\hat{x}_i}), i = N, F$ on the detector states $\hat{x}$ into the Simulink model. The ellipsoid invariants on e are automatically computed by the credible autocoder using equation (18), thus do not need to be expressed on the Simulink model.

C. Prototype Refinements and the Annotations

To automatically transform the semantics of the fault detection and controller system in Figure 3 into useful ACSL annotations, we have further refined the prototype autocoder to be able handle the following issues:

1) Generate different sets of closed-loop semantics based on different assumptions of the plant.
2) Formally expressing the faults to be able to reason about them in the invariant propagation process.

The main change made to the prototype is a new capability to generate multiple different sets of closed-loop semantics based on the assumptions of the different plant semantics. For example, in the generated ACSL annotated code in listing 1, there are two ellipsoid sets parameterized by the the ACSL matrix variables *QMat_1* and *QMat_2*. They express the closed-loop ellipsoid invariant sets $\mathcal{E}(\hat{x}_i, P_{\hat{x}_i}), i = N, F$. The matrix variables are assigned values using the ACSL functions *mat_of_nxn_scalar*, which takes in n^2 number of real-valued arguments and returns an array of size $n \times n$. For brevity's sake, the input arguments to the ACSL functions in listing 1 are truncated. The ellipsoid sets are grouped into two different sets of semantics using the ACSL keyword *behavior*. One set of semantics assumes a nominal plant and the other assumes the faulty one. Each set of semantics are linked to their respective plant models by the behavior name. The pre-conditions, displayed in listing 1 as *requires* statements, are the ellipsoid invariant sets on the observer states $\hat{x}$. The ellipsoid invariants are defined by the function *in_ellipsoidQ*. The post-conditions, which are expressed using the *ensures* keyword, are generated using the invariant propagation process as described in section V-A. The annotation statement *PROOF_TACTIC* is a non-ACSL element that the prototype autocoder generates to assist the automatic verification of the invariants. For example, to formally prove that the post-conditions in listing 1 is true given the pre-conditions, the automatic analyzer knows to apply the affine transformation strategy.

```
1  /*@ logic matrix QMat_1 = mat_of_8x8_scalar(...);
2  ...
3    logic matrix QMat_3 = mat_of_6x6_scalar(...);
4  /*@
5    behavior nominal_ellipsoid:
6    requires in_ellipsoidQ(QMat_3,
7    vect_of_6_scalar(observer_states[0],
8    observer_states[1],observer_states[2],
9    observer_states[3],observer_states[4],
10   observer_states[5]));
11   ensures in_ellipsoidQ(QMat_41,
12   vect_of_12_scalar(observer_states[0]...,
13   _io_->xhat[0]...));
14         @ PROOF_TACTIC (use_strategy (
               AffineEllipsoid));
15
16         behavior faulty_ellipsoid:
17   requires in_ellipsoidQ(QMat_4,
18   vect_of_6_scalar(observer_states[0],
19   observer_states[1],observer_states[2],
20   observer_states[3],observer_states[4],
21   observer_states[5]));
22   ensures in_ellipsoidQ(QMat_42,
23   vect_of_12_scalar(observer_states[0]..._io_->xhat
         [0]..));
24         @ PROOF_TACTIC (use_strategy (
               AffineEllipsoid));
25 */
26 {
27     for (i1 = 0; i1 < 6; i1++) {
28         _io_->xhat[i1] = observer_states[i1];
29     }
30 }
```

Listing 1. ACSL Expressing Multiple Sets of Closed-loop Semantics

The semantics of the plant models are expressed using predefined C functions *faulty_plant* and *nominal_plant* wrapped in the ghost code statements, which are denoted by the *ghost* keyword. Ghost code statements are ACSL statements that are similar to the actual C code in every aspect except they are not executed and they are restricted from changing

the state of any variable in the code. The semantics of the plant model are connected with their respective set of ellipsoid invariants through the use of assertions. For example, in listing 2, we have the plant states *faulty_state* and *nominal_state*, which are declared in the ghost code. They are linked to the same variable *_io_->xp* from the code in both ACSL behaviors using the equal relation symbol ==.

```
/*@ ghost double faulty_state[6]; */
/*@ ghost double nominal_state[6];*/
/*@
        behavior nominal_ellipsoid:
        assumes _io_->xp==nominal_state
        requires in_ellipsoidQ(...);
  ensures in_ellipsoidQ(...);
       @ PROOF_TACTIC (use_strategy (
           AffineEllipsoid));*/
/*@
        behavior faulty_ellipsoid:
  assumes _io_->xp==faulty_state
  requires in_ellipsoidQ(...);
        ensures in_ellipsoidQ(...);
        @ PROOF_TACTIC (use_strategy (
            AffineEllipsoid));*/
{ for (i1 = 0; i1 < 6; i1++) {
  xp[i1] = _io_->xp[i1]; }}
...
/*@ ghost faulty_plant(_io_->u,faulty_plant_state);
  ghost nominal_plant(_io_->u,faulty_plant_state);
      */
```

Listing 2. ACSL Expressing the Semantics of the Plants with C code

Lastly, the autocoder produces the inductive ellipsoid invariants on the error $e = x - \hat{x}$ by first creating the ghost variable array *error_states*, which corresponds to the error $e = x - \hat{x}$, and then inserting the ellipsoid invariants as pre and post-conditions for the ghost code. As shown in listing 3, the autocoder defines the error e using the ghost *for*-loop statement. The ellipsoid invariants, as in the code listings before, are expressed as *require* and *ensure* statements separated into two sets of behaviors that correspond to either the nominal or faulty plant.

```
/*@
  behavior ellipsoid_nominal:
  requires  in_ellipsoidQ(...);
  ensures in_elilpsoid(...);
  behavior ellipsoid_faulty:
  requires in_ellipsoidQ(...);
  ensures in_ellipsoidQ(...); */
{
/*@ ghost for (i1=0; i1<6; i1++) {
    error_states[i1]=x[i1]-observer_states[i1];
  }*/
}
```

Listing 3. ACSL Expressing Invariant Sets on the Error

VI. Conclusion

In this paper, we have presented a framework that can rapidly generate fault detection code with a formal assurance of high-level fault detection semantics such as stability and correct fault detection. The properties are formally expressed using ellipsoid invariants. The framework, named the credible autocoding, can generate the fault detection code as well as the invariant properties for the code. Moreover, the generated invariant properties can be be verified in using semi-automatic theorem prover. We have demonstrated that the credible autocoding prototype that was previously applied to control systems can be extended to fault detection systems with some additions. We applied the prototype tool to an example of observer-based fault detection system running with a LQR controller of a 3 degrees-of-freedom helicopter. The prototype was able to autocode the fault detection semantics successfully. In this paper we only consider a simple output observer for fault detection to demonstrate the autocoding steps. However, the idea can be extended to more complicated fault detection methods.

References

[1] R. Isermann, "Supervision, fault-detection and fault-diagnosis methods—an introduction," *Control engineering practice*, vol. 5, no. 5, pp. 639–652, 1997.

[2] A. Esna Ashari, R. Nikoukhah, and S. L. Campbell, "Auxiliary signal design for robust active fault detection of linear discrete-time systems," *Automatica*, vol. 47, no. 9, pp. 1887–1895, 2011.

[3] H. Niemann and J. Stoustrup, "Design of fault detectors using H_∞ optimization," in *Decision and Control, 2000. Proceedings of the 39th IEEE Conference on*, vol. 5. IEEE, 2000, pp. 4327–4328.

[4] H. H. Niemann and J. Stoustrup, "Robust fault detection in open loop vs. closed loop," in *Decision and Control, 1997., Proceedings of the 36th IEEE Conference on*, vol. 5. IEEE, 1997, pp. 4496–4497.

[5] A. Esna Ashari, R. Nikoukhah, and S. Campbell, "Active robust fault detection in closed-loop systems: quadratic optimization approach," *IEEE Transactions on Automatic Control*, 2012.

[6] S. Ding, *Model-based fault diagnosis techniques: design schemes, algorithms, and tools*. Springer, 2008.

[7] R. Patton and J. Chen, "Robust model-based fault diagnosis for dynamic systems," 1999.

[8] R. Isermann, *Fault-diagnosis systems: an introduction from fault detection to fault tolerance*. Springer, 2005.

[9] T. Wang, R. Jobredeaux, and E. Feron, "A graphical environment to express the semantics of control systems," 2011, arXiv:1108.4048.

[10] E. Feron, R. Jobredeaux, and T. Wang, "Autocoding control software with proofs i: Annotation translation," in *Digital Avionics Systems Conference (DASC), 2011 IEEE/AIAA 30th*, oct. 2011, pp. 1 –19.

[11] H. Herencia-Zapana, R. Jobredeaux, S. Owre, P.-L. Garoche, E. Feron, G. Perez, and P. Ascariz, "Pvs linear algebra libraries for verification of control software algorithms in c/acsl," in *NASA Formal Methods*, 2012, pp. 147–161.

[12] T. Wang, R. Jobredeaux, H. Herencia, P.-L. Garoche, A. Dieumegard, E. Feron, and M. Pantel, "From design to implementation: an automated, credible autocoding chain for control systems," 2013, arXiv:1307.2641.

[13] E. Feron, "From control systems to control software," *Control Systems, IEEE*, vol. 30, no. 6, pp. 50 –71, dec. 2010.

[14] J. Feret, "Static analysis of digital filters," in *European Symposium on Programming (ESOP'04)*, ser. LNCS, no. 2986. Springer-Verlag, 2004.

[15] Q. Quanser Manual, *Quanser 3-DOF Helicopte, user manual*. Quanser Inc, 2011.

[16] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, pp. 576–580, October 1969. [Online]. Available: http://doi.acm.org/10.1145/363235.363259

[17] S. Boyd, L. El Ghaoui, E. Feron, and V. Balakrishnan, *Linear Matrix Inequalities in System and Control Theory*, ser. Studies in Applied Mathematics. Philadelphia, PA: SIAM, June 1994, vol. 15.

[18] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto, *ACSL: ANSI/ISO C Specification Language*, 2008, http://frama-c.cea.fr/acsl.html. [Online]. Available: http://frama-c.cea.fr/acsl.html

[19] E. Feron, "From control systems to control software," *Control Systems, IEEE*, vol. 30, no. 6, pp. 50–71, 2010.

[20] A. B. Kurzhanski and I. Valyi, *Ellipsoidal calculus for estimation and control*, ser. Systems & control. Laxenburg, Austria: IIASA Boston, 1997. [Online]. Available: http://opac.inria.fr/record=b1131065