Chapter 5
From Design to Implementation: An Automated, Credible Autocoding Chain for Control Systems

Timothy Wang, Romain Jobredeaux, Heber Herencia, Pierre-Loïc Garoche, Arnaud Dieumegard, Éric Feron and Marc Pantel

5.1 Introduction

A wide range of today's real-time embedded systems, especially their most critical parts, relies on a control-command computation core. The control-command of an aircraft, a satellite, a car engine, is processed into a global loop repeated during the activity of the controlled device. This loop models the acquisition of new input values via sensors, either from environment mesures (wind speed, acceleration, engine RPM, ...) or from human feedback through, for example, the brakes, the accelerator, the stick or wheel control.

T. Wang (✉) · R. Jobredeaux · É. Feron
Georgia Institute of Technology, Atlanta, GA, USA
e-mail: timothy.wang@gatech.edu

R. Jobredeaux
e-mail: romain.jobredeaux@gatech.edu

É. Feron
e-mail: feron@gatech.edu

H. Herencia
General Electric Global Research, Clifton Park, NY, USA
e-mail: heber.herencia-zapana@ge.com

P.-L. Garoche
ONERA—The French Aerospace Lab, Toulouse, France
e-mail: pierre-loic.garoche@onera.fr

A. Dieumegard · M. Pantel
ENSEEIHT, Toulouse, France
e-mail: arnaud.dieumegard@enseeiht.fr

M. Pantel
e-mail: marc.pantel@enseeiht.fr

E. Feron (ed.), *Advances in Control System Technology for Aerospace Applications*, Lecture Notes in Control and Information Sciences 460, DOI 10.1007/978-3-662-47694-9_5

The cost of failure of such systems is significant, and examples of such failures are numerous, in spite of increasingly high certification requirements. In addition, as the Federal Aviation Administration (FAA) works on defining the proper frame to open the airspace to Unmanned Aerial Systems (UAS), a major market is about to bloom and will benefit from automated and simple tools to facilitate product certification. Current analysis tools focus mainly on simulations. One obvious shortcoming is the impossibility to simulate all the possible scenarios the system will be subject to. More advanced tools include static analysis modules, which derive properties of the system by formally analyzing its semantics. However, in the specific case of control systems, analyzing the computational core can prove arduous for these tools, whereas the engineers who designed the controller have a variety of mathematical results that can greatly facilite this analysis, and evince more subtle properties of the implemented controller.

There are many modern techniques to analyse software. Model checking is one that endeavors to automatically prove safety-properties of finite-state systems [1]. It is widely used in industry as recent developments have made SAT solvers and SMT solvers much more efficient and scalable [2, 3]. Unfortunately, control software remains subject to an explosion of the state-space, making the use of these techniques difficult for this research.

Abstract interpretation has proven to be a powerful, scalable technique to prove low-level properties of code. It was successfully applied on the Airbus A380 code to prove the absence of runtime errors caused by buffer overflow or index out-of-bound failures [4]. The choice of a proper abstract domain and good widening/narrowing heuristics remains a difficult one. In particular, there is no good lattice structure on the domain of ellipsoids, crucial to many results of control theory. Finally, some control systems require highly non-linear Lyapunov functions in their proof of stability, involving transcendental functions that no current domain encompasses, to our knowledge. Feret's work [5, 6] is a practical approach to the problem of extracting quadratic invariants in an abstract interpretation framework. Its goal is to address the need by Astrée [7] to handle the linear filters present in Airbus' real time software. Previous work [8] by Monniaux addressed the same class of systems but not on actual code. Both of these efforts address a strict subset of the systems we consider in this work.

This chapter, following previous efforts aimed at demonstrating how control-systemic domain knowledge can be leveraged for code analysis [9, 10], describes a practical implementation of a fully automated framework, which enables a control theorist to use familiar tools to generate credible code, that is, code delivered with certificates ensuring certain properties will hold for all executions.

The focus is on a specific class of controllers and properties, in order to achieve full automation; it also explores various possible extensions.

The chapter is structured as follows: We first present a high level view of the general framework in Sect. 5.2. We then proceed to describe how control semantics can be expressed at different levels of design, in Sect. 5.3. Section 5.4 describes the translation process by which graphical synchronous languages familiar to the control theorist can be turned into credible code. Section 5.12 demonstrates how a proof of correctness can be automatically extracted from the generated code and its annotations.

5.2 Credible Autocoding Framework

Autocoding is an automated programming process that transforms a system expressed in a high-level modeling language such as Simulink or SCADE into a low-level implementation language such as C. In *credible autocoding*, the code is generated along with mathematically verifiable guarantees of functional correctness. The concept of credible autocoding is analogous to credible compilation in [11]. Both processes generate formally verifiable evidences that the output correctly preserves certain semantics of the input. The evidences can be independently checked for correctness by the certification authorities. Unlike credible compilation of Rinard, the formally verifiable evidences of interest in this research are the high-level functional properties of control systems which include stability, robustness and performance. An alternate approach towards producing guarantees for autocoder is building a formally verified autocoder. In a formally verified autocoder, each block transformations are mathematically proved to be correct. This approach is technically challenging and has yet to be demonstrated to be feasible.

Data-flow modeling languages such as Simulink or SCADE are the default industry choice for Model-based development of safety-critical control systems. Within this framework of software development, systems are build using a language of high-level abstraction in order to facilitate rapid design and prototyping. The source code is then generated automatically from the input model using an automated code generation tool or an autocoder. The trustworthiness of the autocoder has often been questioned in the industry [12]. In a data-flow language environment such as Simulink, there are two major elements: "blocks", and "lines." The blocks are functions that perform some operations on its input(s) and then output the result(s). The lines are directed edges that flow from an origin block's output to a destination block's input. This type of connectivity specifies that the origin output is equivalent to the destination input. The blocks are organized into sets of blocks, forming a library of blocks. Some of the blocks have unpredictable variations in their semantics. For example, the precise semantics of the Simulink product block depends on its input types, input dimensions, number of input/output, product operator selected, etc. The variations in Simulink block semantics, and the lack of formal documentation about them, present an obstacle in its wide adoption for safety-critical production. A related work [13], which is complementary to this research used a model-based approach, to assign provably correct semantics to a set of Simulink blocks. The result of that research is a library of trustworthy blocks—the *BlockLibrary* language—with precise semantics, that can be reasoned about formally.

In the framework of credible autocoding, instead of proving that individual block transformations are correct i.e. building the library of trustworthy blocks, the goal is to be able to show that the output code also satisfies the high-level functional properties of the input model. The functional properties of the input model is dependent on the domain of the input model. In the domain of control systems, a strong functional property is an exponential stability of the closed-loop system with a global domain of attraction, and a weaker functional property is simply the boundedness of the system. In the domain of convex optimization, an example property is the linear convergence of the duality gap function to zero. The verification of the code against high-level functional property imparts an additional layer of guarantee on the correctness of the code. For example, if a gain in a Simulink model was inverted accidentally before autocoding, the output code while correct in the sense of each individual block transformations, is not likely to satisfy a pre-computed property such as the Lyapunov stability of the system.

The complete credible autocoding process from Simulink model to verified code, for the domain of control systems, is illustrated in Fig. 5.1. In this process, the credible autocoding portion (left half of Fig. 5.1) is performed by the code producers, who generate the code and provide evidence that the generated code is correct. The proof-checking portion (right half of Fig. 5.1) are performed by the certification authorities who are independent from the code producers. The only shared knowledge between these two parties is a common language used to express the mathematical proofs on the code. The *control semantics* include stability property of the system and the plant models used in the stability analysis for the closed-loop cases. The framework adds, on top of a classic model-based development cycle, another layer of translations, that converts quadratic invariant sets, computed using Lyapunov-based methods, into code annotations on the code, which form a proof of the correctness of the output code.

For credible autocoding of control software, compared against the traditional model-based development, the only additional requirement on control engineers is

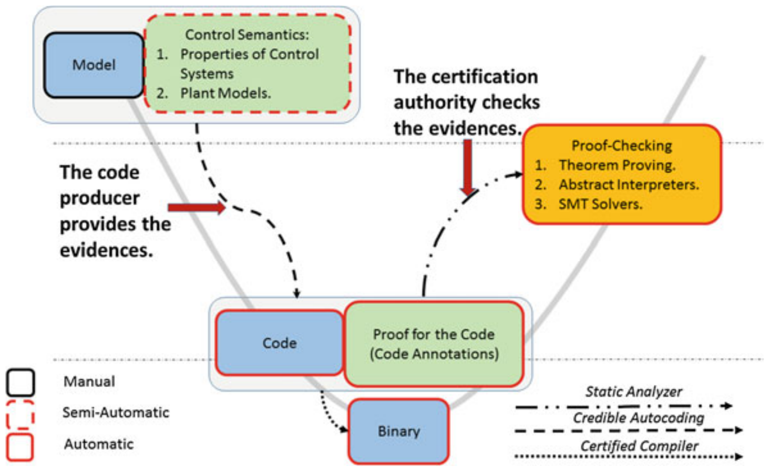
Model
Control Semantics:
1. Properties of Control Systems
2. Plant Models.
The certification authority checks the evidences.
Proof-Checking
1. Theorem Proving.
2. Abstract Interpreters.
3. SMT Solvers.
The code producer provides the evidences.
Code
Proof for the Code (Code Annotations)
Binary
Manual
Semi-Automatic
Automatic
Static Analyzer
Credible Autocoding
Certified Compiler

Fig. 5.1 Automated credible autocoding/compilation chain for control systems

that they provide the Lyapunov function. The procedure for generating Lyapunov-type certificate of stability and performance properties of control systems can be automated using Linear Matrix Inequalities [14] (LMIs) and the Integral Quadratic Constraint [15] (IQC) framework. Each of the stability and performance properties generated can be encoded using an ellipsoid invariant, which can then be transformed into an invariant for the code.

The advantages of the framework developed in this chapter can be summarized as follows:

1. All the advantages of model-based development are inherited.
2. The correctness of the autocoder is more or less guaranteed by the correctness of its output code. This is based on the idea of credible compilation in [11]. This reduces the need to formally verify the autocoder.
3. The framework provides guarantees are of high-level functional properties, which provides a potentially more useful characterization of the correctness of the system to the certification authorities.
4. The framework can provide feedback information to the domain expert so errors in the construction of the model could be diagnosed more rapidly.
5. In the context of certification, credible autocoding could potentially reduce the number of tests required for certification of the control software. In traditional unit testing, a piece of code, such as the control software is tested repeatedly for many possible different inputs and scenarios. This is extremely time consuming. The credible autocoding framework enables a meta-testing procedures, in which the software, are tested for all possible inputs and scenarios, in one iteration.

5.2.1 Input and Output Languages of the Framework

The input language of the framework should be a graphical data-flow modeling language such as Simulink that is familiar to control engineers. The exact choice for the input language is up to the domain users' preference and does not affect the utility of the framework as it can be eventually adapted to other modeling languages such as SCADE [16]. For the prototype tool-chain developed in this research, the choice of the input language is a discrete-time subset of Simulink blocks. The subset of Simulink language include basic blocks: unit delays, gain, input, output, plus, minus, multiplication, divide, min, and max.

Likewise, for the output language, the choice is likely to depend on the preferences of the industry and the certification authority. For the experimental prototype described in this chapter, the output language was chosen to be C because of its industrial popularity and the wide availability of static analyzers tailored for C code.

The set of annotations in the output source code contains both the functional properties inserted by the domain expert and the proofs, which can be used to automatically prove these properties. For the analysis of the annotated output, a prototype annotation checker that is based on the static analyzer frama-C and the theorem prover

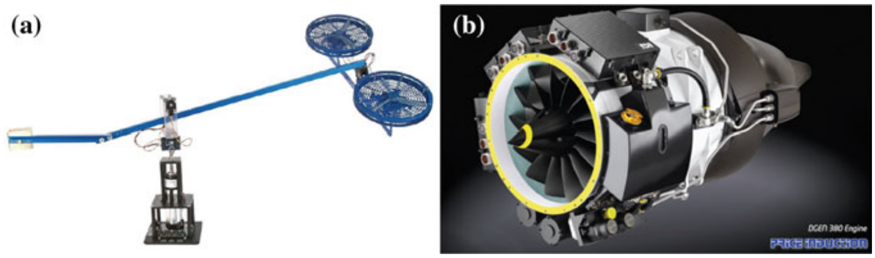

Fig. 5.2 Test platforms. **a** Quanser helicopter (© Quanser), **b** DGEN 380 lightweight turbofan engine (© Price Induction)

PVS is built. For automating the proof-checking of the annotated output, a set of linear algebra definitions and theories were integrated into the standard NASA PVS library [10].

In this chapter, the fully automated process from the input model to the verified output is showcased for the property of close-loop stability. At this point, we restrict the input space to only linear controllers with possible saturations in the loop. The running example that we use in this paper is described by the state-space difference equation in Example 5.1. This example is chosen because it has enough complexity to be representative of many controllers used in the industry, and is simple enough such that we can show in this paper, the output annotated code. The example system is consisted of states $x \in \mathbb{R}^2$, input $y \in \mathbb{R}$, output $u \in \mathbb{R}$, the state-transition function in (5.1), and the output function in (5.2).

Example 5.1

$$x_+ = \begin{bmatrix} 0.4990 & -0.05 \\ 0.01 & 1 \end{bmatrix} x + \begin{bmatrix} 0 \\ 0.01 \end{bmatrix} y \tag{5.1}$$

$$u = \begin{bmatrix} 564.48 & 0 \end{bmatrix} x + \begin{bmatrix} 1280 \end{bmatrix} y. \tag{5.2}$$

In addition to the two dimensional example used in this chapter, we have worked with several larger systems shown in Fig. 5.2, which include the Quanser 3-degree-of-freedom Helicopter, and a FADEC control system of a small twin jet turbofan engine [17]. The state-space size of the engine FADEC controller is 11.

5.3 Control Semantics

The set of control semantics that we can express on the model includes stability and boundedness.

Remark 5.1 The types of systems in which we can express closed-loop stability properties for are not just limited to simple linear systems like Example 5.1. They also include nonlinear systems that can be modeled as linear systems with bounded nonlinearities in the feedback loops.

5.3.1 Control System Stability and Boundedness

A simple linear system such as Example 5.1 is commonly represented using the following state-space formalism. For matrices $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times m}$, $C \in \mathbb{R}^{k\times n}$, and $D \in \mathbb{R}^{k\times m}$, we have

$$\begin{aligned} x_+ &= Ax + By \\ u &= Cx + Dy \end{aligned} \tag{5.3}$$

This state-space system has the state-transition function $\delta : (x, y) \to Ax + By$ and the output function $\omega : (x, y) \to Cx + Dy$. We also consider linear systems with bounded nonlinearities in their feedback interconnections. This model is a closer representation of the real control systems when there are saturations, time-delays, anti-windup mechanisms, hysteresis, or noise in the loop. For $\tilde{y} = \sigma(t, y)$ with the time-varying nonlinear function $\sigma(t, y)$, in which $\tilde{y}_i \leq M_i y$ for $M_i > 0$, we have

$$\begin{aligned} x_+ &= Ax + B\tilde{y} \\ u &= Cx + D\tilde{y} \end{aligned} \tag{5.4}$$

Analogous to system (5.3), the state-transition function for (5.4) is $\delta : x, y \to Ax + B\sigma(t, y)$, and the output function is $\omega : (x, y) \to Cx + D\sigma(t, y)$. For any systems described by (5.3) and (5.4), we can compute efficiently the answer to the following problem.

Problem 5.1 1. Does there exist a symmetric matrix $P \in \mathbb{R}^{n\times n}$ such that the quadratic function $q : x \to x^{\mathrm{T}}Px$ is non-increasing along the system trajectories as $t \to +\infty$?

Generally speaking such a problem can be transformed into a linear matrix inequality (LMI). The details of these transformations are skipped here as they are well-established in the control literature [14, 18]. The algorithms used to solve LMIs are based on semi-definite programming, which is tractable up to large sizes [19].

5.3.2 Prototype Tool-Chain

In this chapter, we describe a prototype tool-chain, which has been developed for the demonstration of credible autocoding. The prototype tool-chain is split into a credible autocoder front-end and a proof-checker back-end. The credible autocoder front-end translates the model into annotated code. The proof-checking back-end analyzes the annotated code produce by the front-end and decides whether or not the proof is coherent.

Gene-Auto [20–23] is an existing, open-source, autocoding prototype for embedded systems. The front-end prototype in our tool-chain, which we dubbed Gene-Auto+, is based on Gene-Auto. For the front-end, we have several extensions to the language or language environments Simulink, Gene-Auto and ACSL. ACSL [24] is a formal specification language for C programs. More details on ACSL is described in Sect. 5.3.7. The language extensions are summarized as follows:

1. A library of Annotation blocks in Simulink and Gene-Auto.
2. An ACSL-like language within Gene-Auto.
3. Abstract types and their operators in ACSL: matrix, vector and quadratic predicates.

The language extensions in Simulink and Gene-Auto are described in Sect. 5.3.3. The language extensions in ACSL are described in Sect. 5.3.7.

5.3.3 Control Semantics in Simulink and Gene-Auto

An observer (see [25]) in Simulink takes an input signal and returns an output of 1 if the input satisfies a specific property, and 0 if otherwise. Both boundedness and stability can be expressed, for example, using an observer with inputs x_i, $i = 1, \ldots, n$, and the boolean-valued function

$$x \to \sum_{i,j=1,\ldots,n} x_i P_{ij} x_j \leq 1. \tag{5.5}$$

To express the types of observers in (5.5) as annotations on the Simulink model, we extended the Simulink language and the Gene-Auto environment with a set of annotation blocks.

Annotation blocks are structurally the same as any other Simulink blocks. The key difference is that they do not translate into code. Our prototype annotation block library has been built to contain a minimal set of blocks needed to express the properties of control systems that are currently verifiable from Simulink to C code.

The prototype annotation block library contains four symbols: *vamux*, *constant*, *quadratic*, and *system*. Each annotation symbol denotes an annotation block type, To illustrate the annotations blocks, we have Fig. 5.3, which shows a Simulink model of an engine controller, along with 6 annotation blocks. The annotation blocks are highlighted in red for the purpose of clarity.

In Simulink, the *vamux* block type takes n scalar or vector inputs x_i, and outputs a concatenated signal $y = \left[x_1^\mathsf{T}, \ldots, x_n^\mathsf{T}\right]^\mathsf{T}$. In Fig. 5.3, there are three *vamux* blocks, labeled as *nh*, $xc(t)$ and $yd(t)$. The *vamux* block type only accepts one parameter, which determines the number of inputs to the block type. The *vamux* block does not express any property of the system. In Gene-Auto+, the main functionality of the

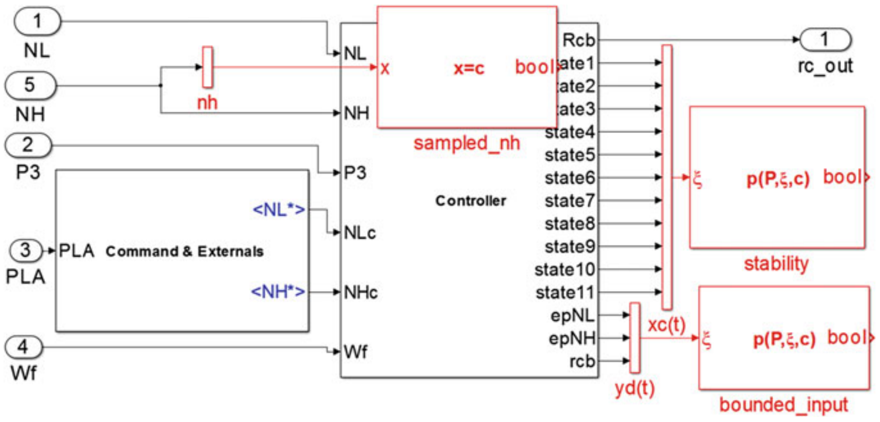

Fig. 5.3 Simulink model with annotation blocks

vamux block is to establish equivalence relations between its inputs and the ith entry of its output. i.e. $x_i == y_i$. This enables the prototype to replace the pseudo-variables in the templates within the other annotation blocks with the actual variables from the code.

The *constant* block type accepts one scalar, vector, or matrix input x, and a constant matrix parameter $[c_1]$ or $[c_1, \ldots, c_n]$ for $n \in \mathbb{N}$. The type of the constants c_i are constrained to be the same type as the input x. The output of the block is the boolean value $x == c_1$ or $\bigvee_{i=1}^{n} (x == c_i)$, which implies n sets of *behaviors* for the code.

Definition 5.1 A behavior is a set of unique assumptions on the parameters and input, and output of the model.

This block type is useful for expressing the semantics of parameter varying systems such as a gain-scheduled controller. For example, the scheduling parameter of the controller in Fig. 5.3 is the input *NH*, which is annotated with a *constant* block labeled *sampled_nh*. In Gene-Auto+, the *constant* block type generates a set of assumption(s) $\{x == c_i\}$, $i = 1, \ldots, n$.

The *quadratic* block type accepts one input vector of n variables x, a matrix parameter $P \in \mathbb{S}^{n\times n}$, a logic connective symbol $\diamond \in \{<=, <, >, ==\}$, a level-set constant $c \in \mathbb{R}$, and outputs the boolean value of $x^{\mathsf{T}} P x \diamond c$. The *quadratic* block type can be used, for example, to express ellipsoidal invariant sets, sector-bound inequalities, 2-norm squared, sum of squares polynomial sets, etc. The *quadratic* block also accepts a positive scalar parameter *mu*. This is used to indicate the multiplier computed in stability analysis. The *quadratic* block type behaves like an ellipsoid observer from (5.5) in Simulink. In Gene-Auto+, the *quadratic* block type generates a predicate defined on its inputs: $\forall x. x^{\mathsf{T}} P x \leq c$. For example, in Fig. 5.3, the *quadratic* block labeled *stability* represents a claim that $xc(t)$ does not violate a quadratic constraint. The other *quadratic* block *bounded_input* is used to express a bound on the input $yd(t)$ to the controller.

The *system* block type is parameterized by 4 matrices A, B, C, and D. An example of a Simulink model annotated with the *system* block can be found in Sect. 5.3.5. The *system* block type accepts two vector inputs u and y. The output of the *system* block type is the state x of the dynamical system

$$\begin{aligned} x_+ &= Ax + Bu, \quad x(0) = x_0 \\ y &= Cx + Du. \end{aligned} \tag{5.6}$$

The semantics of the *system* block in Gene-Auto include the semantics of the discrete-time linear state-space system in (5.6), and a set of relations $\{\tilde{y}_i == y_i, u_i = \tilde{u}_i\}$ that establish equivalence between the annotation variables y and u and their corresponding variables $\tilde{y}$ and $\tilde{u}$ from the controller model. The *system* block type can be used, for example, to express a model of the plant the controller is expected to interact with. The same controller model can be annotated with multiple *system* blocks, which results in multiple sets of predicates for the code, which can be annotated using the *behavior* keyword from ACSL.

5.3.4 Annotation Blocks and Behaviors in the Model

In a model, multiple *system* blocks $s_1, \ldots, s_n$ can be connected to the same set of *vamux* blocks. This results in a set of n behaviors expressed by the formula $\bigvee_i^n s_i$. If there are n *system* blocks connected to the controller model, then there are n behaviors in the model.

If there are also k *constant* blocks in the model, each connected to a different *vamux* block, and each with m behaviors, then we have a total of m^k behaviors resulting from the *constant* blocks: $\bigwedge_i^k \left(\bigvee_i^m c_i\right)$. The complete set of behaviors in the model resulting from both the *system* and *constant* blocks is described by the formula

$$\left(\bigwedge_i^k \left(\bigvee_i^m c_i\right)\right) \wedge \left(\bigvee_i^n s_i\right) \tag{5.7}$$

or a total of nm^k possible behaviors.

Lastly, if there are w *quadratic* blocks in the model as well, and all of them are connected to the same set of *vamux* block, then we have w number of behaviors $\vee_i^w q_i$ due to the *quadratic* blocks. Combining this set of behaviors conjunctively with the set of behaviors generated by the *system* and *constant* blocks results in

$$\left(\bigwedge_i^k \left(\bigvee_i^m c_i\right)\right) \wedge \left(\bigvee_i^n s_i\right) \wedge \left(\bigvee_i^w q_i\right) \tag{5.8}$$

for a possible total of wnm^k behaviors in the model. However, each of the *quadratic* blocks that encode an inductive property such as stability are typically assigned a behavior generated by a *system* block. This is true for many examples, in which the quadratic invariant is computed based on some plant model, using independent LMI-based tools. For example, if there are n quadratic invariants and each is assigned a behavior from a *system* block, then there are only n behaviors in the model:

$$\bigvee_{i}^{n} (s_i \wedge q_i) \, . \tag{5.9}$$

This is far less than the explosion in the number of of behaviors predicted by (5.8).

Next, some annotated examples are given. Each example contains a different possible set of control semantics.

5.3.5 Closed-Loop Stability with Bounded Input

For the running example, the closed-loop stability of the system with bounded input is expressed with a set of the *system* and *quadratic* blocks. For example, as displayed in Fig. 5.4, the close-loop stability of the Simulink model of the control systems is expressed using:

1. a *quadratic* block *stability* to express the ellipsoidal invariant set that encodes the closed-loop stability of the system.
2. another *quadratic* block *bounded_input* to express a 2-norm bound on the input,
3. a *system* block *plant*, which expresses the discrete-time linear state-space system used in the closed-loop stability analysis

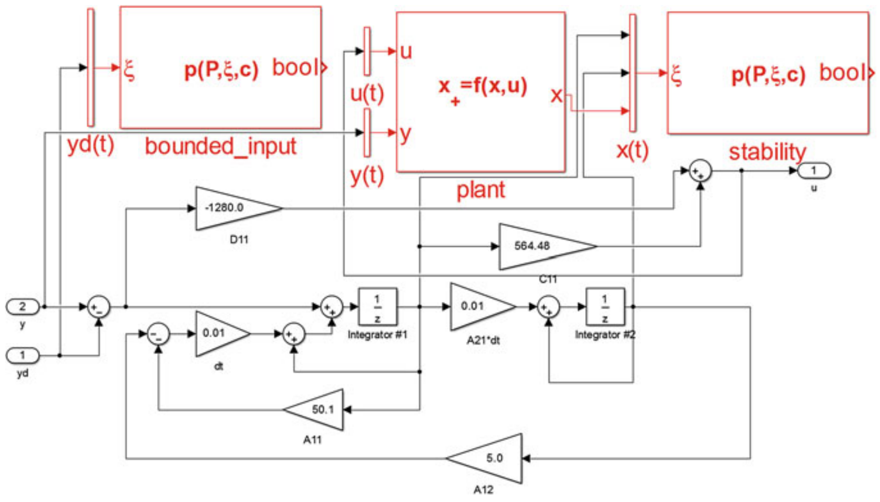

Fig. 5.4 Control system model annotated with control semantics

5.3.6 Expressing the Observer-Based Fault-Detection Semantics

In an observer-based fault-detection system, the dynamics of the observer are designed such that the output of the observer changes due to specific faults in the plant. Once the change exceeds a certain pre-defined threshold, the system is said to be in the faulty mode. To express the faulty and nominal behavior of a fault-detection system, one can use two different *system* blocks. One *system* block is the model of the faulty plant that is predicted to trigger the faulty mode and the other is the nominal plant. This is displayed in Fig. 5.5. The *quadratic* blocks connected to the *vamux* blocks $xf(t)$ and $xn(t)$ express the closed-loop stability of the system. They are assigned behaviors based on their physical connections to the *system* block. For example, as displayed in Fig. 5.5, the block *cl_faulty* is connected to the *system* block *quanser_faulty* using the *vamux* block $xf(t)$. The two *quadratic* blocks

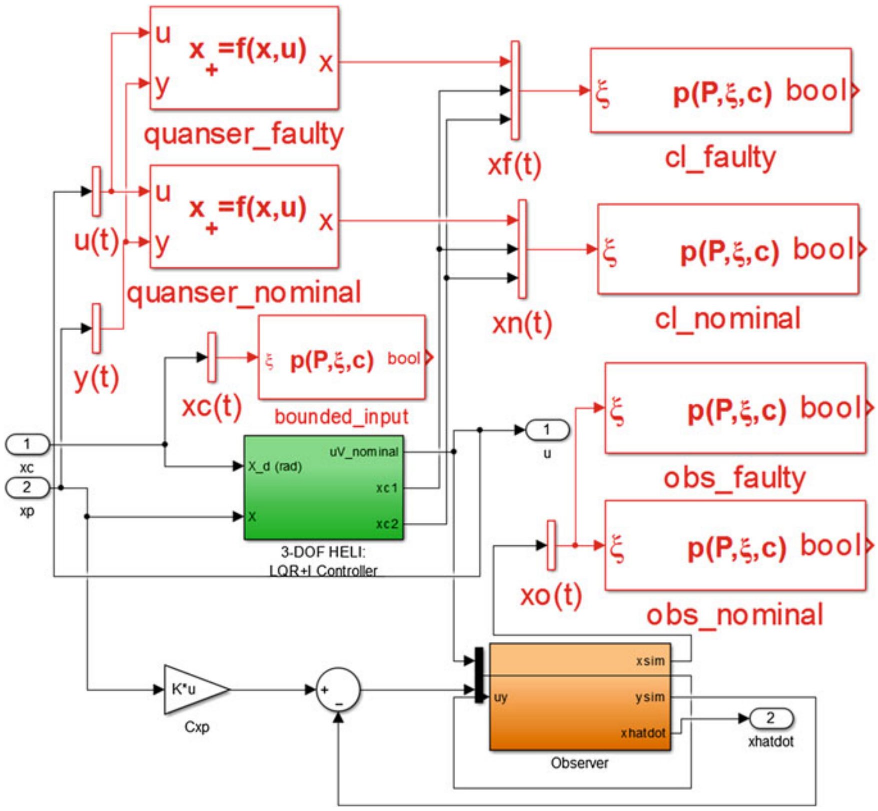

Fig. 5.5 Expressing multiple behaviors: fault-detection system

connected to the *vamux* block $xo(t)$ are used to express the stability of the observer dynamics. They are assigned the behaviors *faulty* and *nominal*, based on the labels in their names.

5.3.7 Control Semantics at the Level of the C Code

For the specific problem of open loop stability, the expressiveness needed at the C code level is twofold. On the one hand, one needs to express that a vector composed of program variables belongs to an ellipsoid. This entails a number of underlying linear algebra concepts. On the other hand, one needs to provide the static analysis tools with indications on how to proceed with the proof of correctness.

The ANSI/ISO C Specification Language (ACSL), is an annotation language for C [24]. It is expressive enough to fulfill our needs, and its associated verification tool, Frama-C [26], offers a wide variety of back-end provers that can be used to establish the correctness of the annotated code.

5.3.7.1 Linear Algebra in ACSL

A library of ACSL symbols has been developed to express concepts and properties pertaining to linear algebra. In particular, types have been defined for matrices and vectors, and predicates expressing that a vector of variables is a member of the ellipsoid $\mathscr{E}_P$ defined by $\{x \in \mathbb{R}^n : x^{\mathrm{T}}Px \leq 1\}$, or the ellipsoid $\mathscr{G}_X$ defined by $\left\{x \in \mathbb{R}^n : \begin{bmatrix} 1 & x^{\mathrm{T}} \\ x & X \end{bmatrix} \geq 0\right\}$. For example, expressing that the vector composed of program variables v_1 and v_2 is in the set $\mathscr{E}_P$ where $P = \begin{pmatrix} 1.53 & 10.0 \\ 10.0 & 507 \end{pmatrix}$, can be done with our ACSL extensions using the annotations in Fig. 5.6.

The invariance of ellipsoid $\mathscr{E}_P$ throughout any program execution can be expressed by the *loop invariant* in Fig. 5.7. This annotation expresses that before and after every execution of the loop, the property $\begin{bmatrix} v_1 & v_2 \end{bmatrix}^{\mathrm{T}} \in \mathscr{E}_P$ will hold. In terms of expressiveness, it is all that is required to express open loop stability of a linear controller. However, in order to facilitate the proof, intermediate annotations are added within the loop to propagate the ellipsoid through the different variable assignments, as suggested in [9] and expanded on in Sect. 5.4. For this reason, a loop body instruction can be annotated with a local contract, as in Fig. 5.8.

```
/*@ logic matrix P = mat_of_2x2_scalar(1.53,10.0,10.0,507);
  @ assert in_ellipsoid(P,vect_of_2_scalar(v_1,v_2)); */
```

Fig. 5.6 Asserting that a vector with components v_1 and v_2 belongs to ellipsoid $\mathscr{E}_P$ in ACSL

```
//@ loop invariant in_ellipsoid(P,
//@                                vect_of_2_scalar(v_1,v_2));
while (true){
  //loop body
}
```

Fig. 5.7 Expressing the invariance of $\mathscr{E}_P$ on a loop in ACSL

```
/*@ requires in_ellipsoid(P,vect_of_2_scalar(v_1,v_2));
  @ ensures in_ellipsoid(Q,vect_of_3_scalar(v_1,v_2,v_3));*/
{
// assignment of v_3
}
```

Fig. 5.8 A local contract to assist the proof process

```
/*@ requires in_ellipsoid(P,vect_of_2_scalar(v_1,v_2));
  @ ensures in_ellipsoid(Q,vect_of_3_scalar(v_1,v_2,v_3));
  @ PROOF_TACTIC (use_strategy (AffineEllipsoid));*/
{
// assignment of v_3
}
```

Fig. 5.9 Adding proof tactics to a contract to guide the proof back-end

5.3.7.2 Including Proof Elements

An extension to ACSL, as well as a plugin to Frama-C, have been developed. They make it possible to indicate the proof steps needed to show the correctness of a contract, by adding extra annotations. For example, the syntax in Fig. 5.9 signals Frama-C to use the strategy `AffineEllipsoid` to prove the correctness of the local contract considered. Section 5.12 expands on this topic.

5.3.8 Closed Loop Semantics

In order to express properties pertaining to the closed loop behavior of the system, one needs to introduce a model for the plant, to be able to refer to the plant variables. The most accurate way to do so would require a hybrid system representation, given that the plant is commonly a continuous system, while the digital controller is a discrete one. A large body of work is devoted to proving meaningful properties of hybrid systems. In order to obtain actionable results, on which proof can be carried out, we made the choice of representing the plant as a linear system, discretized at the same period as the controller. To achieve this, we use ACSL's ghost code feature.

Ghost code is a way to introduce variables and operations on these variables without affecting the semantics of the code. Any valid C code can be written in ghost code as long it does not introduce a change in the actual variables.

At the end of the control loop, we use these variables to express the state update of the plant that results from the computed control signal value. We also enforce axiomatically the fact that the input read from the sensors equals the output of the plant.

For each state variable in the plant, we introduce a global ghost variable. Within the update function of the controller, we introduce ghost code describing the state update resulting from the control output. A template of the structure of the code is given in Fig. 5.10.

5.3.9 *Control Semantics in PVS*

Through a process described in Sect. 5.12, verifying the correctness of the annotated C code is done with the help of the interactive theorem prover PVS. This type of prover normally relies on a human in the loop to provide the basic steps required to prove a theorem. In order to reason about control systems, linear algebra theories have been developed. General properties of vectors and matrices, as well as theorems specific to this endeavor have been written and proven manually within the PVS environment [10].

5.3.9.1 Basic Types and Theories

Introduced in [10] and available online[1] as part of the larger NASA PVS library, the PVS linear algebra library allows one to reason about matrix and vector quantities, by defining relevant types, operators and predicates, and proving major properties. To name a few, we have defined:

- A vector type.
- A matrix type, along with all operations relative to the algebra of matrices.
- Various matrix subtypes such as square, symmetric and positive definite matrices.
- Block matrices
- Determinants
- High level results such as the link between Schur's complement and positive definiteness

5.3.9.2 Theorems Specific to Control Theory

In [10], a theorem was introduced, named the ellipsoid theorem. A stronger version of this theorem, along with a couple other useful results in proving open loop stability of a controller, have been added to the library. The theorem in Fig. 5.11 expresses in

[1] http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/.

```
/*@     ghost REAL xp_0;
  @     ghost REAL xp_1;
  @     ...
  @     ghost REAL xp_0_tmp;
  @     ghost REAL xp_1_tmp;
  @     ...*/
/*@ requires in_ellipsoidQ(Q,vect_of_n_scalar(xc[0],
                                            xc[1],...,xp_0,xp_1
                                                ,...));
  @ ensures in_ellipsoidQ(Q,vect_of_n_scalar(xc[0],
                                            xc[1],...,xp_0,xp_1
                                                ,...)); */
void update_fun(t_example_io *_io_, t_example_state *xc){
...
/*@ requires pre_i
  @ ensures post_i
  @ PROOF_TACTIC (use_strategy ( strategy_i ) )*/
{
instruction i;
}
...
/*@         behavior Plant_N:
            requires in_ellipsoidQ(QMat_N,vect_of_m_scalar(xc[0],
                xc[1],
                                          ...,xp_0,xp_1,...,u));
            ensures in_ellipsoidQ(QMat_{N+1},vect_of_n_scalar(xc
                [0],xc[1],
                                             ...,xp_0,xp_1,...));
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));*/
{
    /*@
            ghost xp_0_tmp = xp_0;;
            ghost xp_1_tmp = xp_1;;
            ... */
    /*@
            ghost xp_0 = a_11 * xp_0_tmp + a_12*xp_1_tmp +.. + b1
                *u;;
            ghost xp_1 = a_21 * xp_0_tmp + a_22*xp_1_tmp +.. + b2
                *u;;
            ...*/
}
    /*@ behavior Plant_{N+1}:
            requires in_ellipsoidQ(QMat_{N+1},vect_of_n_scalar(xc
                [0],xc[1],
                                              ...,xp_0,xp_1,...))
                                                  ;
            ensures in_ellipsoidQ(Q,vect_of_n_scalar(xc[0],xc[1],
                                    ...,xp_0,xp_1,...));
            @ PROOF_TACTIC (use_strategy (PosDef));
    */
{
}
}
```

Fig. 5.10 Template of the update function with added plant semantics in ghost code. Note that often, the ghost code and the annotations are much larger than the code actually executed

```
ellipsoid_general: THEOREM
 ∀ (n:posnat,m:posnat, Q:SquareMat(n),
            M: Mat(m,n), x:Vector[n], y:Vector[m]):
                in_ellipsoid_Q?(n,Q,x)
                AND y = M*x
         IMPLIES
         in_ellipsoid_Q?(m,M*Q*transpose(M),y)
```

Fig. 5.11 Affine ellipsoid transformation theorem in PVS

```
ellipsoid_combination: THEOREM
 ∀ (n,m:posnat, lambda_1, lambda_2: posreal, Q_1: Mat(n,n),
        Q_2: Mat(m,m), x:Vector[n], y:Vector[m], z:Vector[m+n]):
              in_ellipsoid_Q?(n,Q_1,x)
              AND in_ellipsoid_Q?(m,Q_2,y)
              AND lambda_1+ lambda_2 <= 1
              AND z = Block2V(V2Block(n,m)(x,y))
   IMPLIES
   in_ellipsoid_Q?(n+m,Block2M(M2Block(n,m,n,m)(1/lambda_1*Q_1,
                  Zero_mat(m,n),Zero_mat(n,m),1/lambda_2*Q_2)),z)
```

Fig. 5.12 Ellipsoid combination through S procedure theorem in PVS

the PVS syntax how a generic ellipsoid $\mathscr{G}_Q$ is transformed into $\mathscr{G}_{MQM^T}$ by the linear mapping $x \mapsto Mx$.

The theorem in Fig. 5.12: expresses how, given 2 vectors x and y in 2 ellipsoids $\mathscr{G}_{Q_1}$ and $\mathscr{G}_{Q_2}$, and multipliers $\lambda_1, \lambda_2 > 0$, such that $\lambda_1 + \lambda_2 \leq 1$, it can always be said that $\begin{pmatrix} x \\ y \end{pmatrix} \in \mathscr{G}_Q$, where $Q = \begin{pmatrix} \frac{Q_1}{\lambda_1} & 0 \\ 0 & \frac{Q_2}{\lambda_2} \end{pmatrix}$

These 2 theorems are used heavily in Sect. 5.12 to prove the correctness of a given Hoare triple. While they are not particularly novel, their proof in PVS was no trivial process and required close to 10000 manual proof steps from the authors.

5.4 Autocoding with Control Semantics

The translation process in the credible autocoding prototype Gene-Auto+, is now described in more details with a demonstration on the running example. From the input model to the verified output, the property of open-loop and closed-loop stability for a linear system with a nonlinear, but bounded input is expressed.

5.5 Building the Input Model

The model with the closed-loop stability semantics is already displayed in Fig. 5.4. The model expressing open-loop stability is displayed in Fig. 5.13.

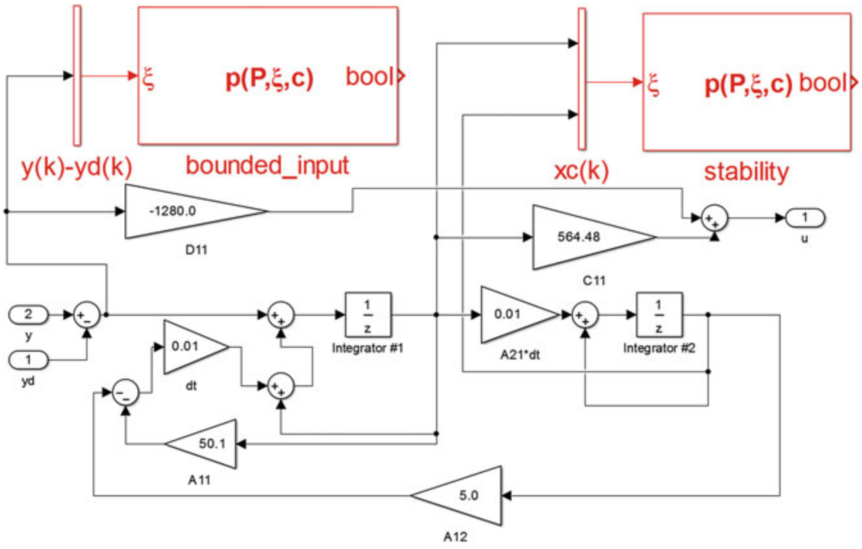

Fig. 5.13 Open-loop stability

In either case, an assumption of boundedness is made on the input to the model and it is expressed by the *quadratic* block *bounded_input*. For the closed-loop case, the assumption of boundedness is made on the signal y_d (see Fig. 5.4). For the open-loop case, a similar boundedness assumption is made on the signal $y - y_d$ (see Fig. 5.13). The closed-loop quadratic invariant, expressing stability, is defined by the multiplier $mu = 0.991$, and $P \succ 0$,

$$P = \begin{bmatrix} 0.1878 & 0.1258 & -0.0813 & 0.0149 \\ 0.1258 & 0.3757 & -0.0220 & 0.0100 \\ -0.0813 & -0.0220 & 0.0660 & -0.0063 \\ 0.0149 & 0.0100 & -0.0063 & 0.0012 \end{bmatrix}. \tag{5.10}$$

Likewise, the stability analysis is also done for the open-loop case. The quadratic invariants are inserted into their respective Simulink model using *quadratic* blocks. Both of them are labeled as *stability*.

5.6 Basics of Program Verification

In the translation process, we use several notions from formal program verification. First we have the following predicate notations for the annotations expressed in this section. The ellipsoid sets are denoted using one of the following symbols:

$$p(P,x,c) \triangleq \left\{x \in \mathbb{R}^n \mid x^{\mathsf{T}} P x \le c\right\}$$
$$q(Q,x,c) \triangleq \left\{x \in \mathbb{R}^n \mid \begin{bmatrix} c & x^{\mathsf{T}} \\ x & Q \end{bmatrix} \succ 0\right\}. \tag{5.11}$$

Without loss of generality, the sublevel set parameter c is set to 1 unless described otherwise.

The control semantics are translated into axiomatic semantics on the code. Axiomatic semantics is one of several approaches in theoretical computer science to assign mathematical meanings to a program [27]. In axiomatic semantics, the semantics or mathematical meanings of a program are defined using the logic predicates that hold before the execution of the code and the ones that hold the execution of the code. The main structure of axiomatic semantics is a *Hoare triple* [28].

Definition 5.2 A Hoare triple is the 3-tuple $(\{P\}, C, \{Q\})$, in which P is a predicate or a set defined by a formula in some logic, and Q is also another predicate, and C denotes a block of code.

The symbol P denotes a post-condition and the symbol Q denotes a pre-condition.

Definition 5.3 A Hoare triple $\{P\}, C, \{Q\}$ is interpreted to be *partially correct*, if P holds before the execution of C, and Q holds after the execution of C.

Remark 1 The termination of C needs to be proved for correctness. For the rest of this chapter, correctness refers to the notion of partial correctness.

The pre and post-conditions are expressed on the code as comments before and after the block of code. For example, given the simple `while` program in Fig. 5.14, If the statement $|x| \le 1$ holds before the execution of the loop, then it should hold for all executions of the loop.

Definition 5.4 An *invariant* is a predicate that holds for all executions of the loop.

The statement $|x| \le 1$ is an invariant. It can be inserted into the code as both the pre-condition and the post-condition, see the ACSL comments in Fig. 5.14. The Hoare triple in Fig. 5.14, therefore is $\{|x| \le 1\}$ `while` a do C end $\{|x| \le 1\}$.

Here we give an illustration of Hoare triples on a Matlab implementation of $x_+ = Ax+By$ (see Fig. 5.15). A Matlab example is used in this section and further on only for the sake of brevity and clarity. In practice, a language like C is the typical choice for the implementation of real-time control systems. We assume the Matlab example satisfies

```
// abs(x)<=1;
while (x*x>0.5)  {
  x=0.9*x;
}
// abs(x)<=1;
```

Fig. 5.14 A `while` program in C

```
% x'*P*x<=1;
while (t<5000)
  u=C*x+D*y;
  x=A*x+B*y;
end
% x'*P*x<=1;
```

Fig. 5.15 Annotated lead/lag compensator in matlab

some ellipsoidal invariant $p(P, x, 1)$, computed from a stability analysis [14]. The invariance of $p(P, x, 1)$, which is the property of interest, is translated into axiomatic semantics for the Matlab code. This is done by translating the set $q(P, x, 1)$, using type matching, into the Matlab formula `x*P*x<=1` and then inserting that formula as pre and post-condition for the program. The result is the annotated Matlab program in Fig. 5.15. Next, the basics of deductive program verification are described.

5.6.1 Hoare Logic and Deductive Verification

Hoare logic is a formal proof system that comes with a set of axioms and inference rules for reasoning about the correctness of Hoare triples on various structures of an imperative programming language i.e. `if-else` statements, `assignment` statements, `while` statements, `for` statements, *empty* statements, etc.

For example, an axiom in Hoare logic for the `while` program construct is

$$\frac{\{P \wedge a\}\ C;\ \{P\}}{\{P\}\ \texttt{while}\ a\ \text{do}\ C\ \text{end}\ \ \{\neg a \wedge P\}}. \tag{5.12}$$

Syntactically speaking, the axioms and inference rules can be interpreted as follows: the formula above the horizontal line implies the formula below that line. In the `while` axiom in (5.12), note that pre and post-conditions of the loop has to be same formula. This means to verify program loops, an invariant is necessary. Some of the basic inferences rules for reasoning about imperative programs using Hoare logic are listed in Table 5.1. The consequence rule in (5.13) is useful whenever a stronger pre-condition or weaker post-condition is needed. The term stronger here means the set defined by the predicate is smaller. The term weaker means precisely the opposite. The substitution rule in (5.16) are used when the code is an *assignment* statement. The weakest pre-condition expression $P[x/expr]$ in (5.16) means P with all free occurrences of the expression *expr* replaced by x. For example, given a post-condition `y<=1` for the line of code `y=x+1`, one can deduct that `x+1<=1` is a weakest pre-condition using the backward substitution rule in (5.16). The skip rule in (5.15) can be used when the executing piece of code does not change any variables in the pre and post-conditions.

Table 5.1 Hoare logic inference rules for a imperative language

$$\frac{\{P_1 \implies P_2\}C\{Q_1 \implies Q_2\}}{\{P_1\}C\{Q_2\}} \quad (5.13) \qquad \frac{\{P\}C_1\{R\};\{R\}C_2\{Q\}}{\{P\}C_1;C_2\{Q\}} \quad (5.14)$$

$$\frac{}{\{P\}SKIP\{P\}} \quad (5.15) \qquad \frac{}{\{P[e/x]\}x{:=}expr\{P\}} \quad (5.16)$$

1. $\{p(P,x,1)\}$ `while` a do C end$\{p(P,x,1)\}$.
2. $\{p(P,x,1)\}C\{p(P,x,1)\}$ by the `while` axiom in (5.12).
3. $\{p(P,A*x+B*y,1)\}x=A*x+B*y\{p(P,x,1)\}$ by the backward substitution rule in (5.16).
4. $\{p(P,A*x+B*y,1)\}u=C*x+B*y\{p(P,Ax+By,1)\}$ by the skip rule in (5.15).
5. $\{p(P,x,1),p(P,A*x+B*y,1)\}C\{p(P,x,1)\}$ by the composition rule in (5.14).
6. if $p(P,x,1) \implies p(P,A*x+B*y,1)$, then $\{p(P,x,1)\}C\{p(P,x,1)\}$ by the consequent rule in (5.13).

Fig. 5.16 Correctness of the program using Hoare logic deduction

To verify the Hoare triple in Fig. 5.15, use the inference rules from Table 5.1 on the code, starting from the post-condition `x*P*x<=1`.The process produces an alternate pre-condition $q(P, A*x+B*y, 1)$ for the loop body. By the consequent rule, the correctness of the initial Hoare triple can be checked by checking if $p(P, x, 1) \implies p(P, A*x+B*y, 1)$. The process in Fig. 5.16 is deductive. An algorithmic reformulation of it is Dijstra's work on Predicate transformers [29]. By using the Predicate transformers, the deductive process of Fig. 5.16 is reduced to a computational process that checks the correctness of first order formulas.

5.6.2 Predicate Transformers

The Hoare triples on the code are computed using a form of the *weakest pre-condition* calculus. The weakest pre-condition of C is a function wp that maps any post-condition Q to a pre-condition. The output of the weakest pre-condition function $wp(C, Q)$ is the largest set such that, after the execution of C, Q holds. For example, the correctness of a Hoare triple, for a set of variables x in the code C, is determined by checking if the logic formula $\forall x, P \implies wp(C, Q)$ holds. The wp function can be applied to various constructs in an imperative programming language. Some examples are given in Table 5.2. The sequence of I_i in (5.20) can be replaced by a single I if I is an invariant of the loop. Denote the `while` program as $\mathscr{P}$, in the case of partial correctness, $wp(\mathscr{P}, Q) = I$ is the *weakest literal pre-condition* if $I \implies wp(C, I)$. In the case of total correctness, $wp(\mathscr{P}, I) = I$ is the weakest pre-condition, if $I \implies Q$ and the loop terminates. Recall the control program in Matlab from 5.15, which is comprised of a while loop and satisfies the invariant $p(P, x, 1)$, Apply wp-calculus to that program i.e. $wp(\mathscr{P}, p(P, x, 1)) = p(P, x, 1)$ leads to two logic formulas:

Table 5.2 Weakest Pre-condition Calculus

$$wp(C_1;, ..., C_N, Q) = wp(C_1, wp(C_2, wp(C_3, ..., wp(C_N, Q))\dots) \tag{5.17}$$

$$wp(\textbf{skip}, Q) = Q \tag{5.18}$$

$$wp(x := e, Q) = Q[e/x] \tag{5.19}$$

$$\begin{aligned} &wp(\texttt{while}\ a \text{ do } C \text{ end}, Q) = \forall i \in \mathbb{N}, I_i \\ &I_0 = true \\ &I_{i+1} = (\neg a \implies Q) \wedge (a \implies wp(C, I_i)) \end{aligned} \tag{5.20}$$

1. $I \implies Q$ and the loop terminates. The loop in 5.15 terminates after a finite amount of iterations and clearly $p(P, x, 1) \implies p(P, x, 1)$ is true.
2. $I \implies wp(C, I)$ i.e. $p(P, x, 1) \implies wp(C, p(P, x, 1))$.

The second condition is harder to verify since the set $wp(c, p(P, x, 1))$ need to be computed. Notice the formula $p(P, x, 1) \implies wp(C, p(P, x, 1))$ is equivalent to the Hoare triple

$$\{p(P, x, 1), wp(C, p(P, x, 1))\, C; \{p(P, x, 1)\}, \tag{5.21}$$

which means that $p(P, x, 1)$ can be inserted as the pre and post-conditions of the loop body C in Fig. 5.15. Applied additional wp-calculus on the loop body results in the annotated code in 5.17. The set of pre-conditions generated by wp-calculus i.e. the displayed Matlab comments inside the loop in Fig. 5.17, along with the Matlab code itself, forms the translated proof of stability. Verifying this proof implies that $q(P, x, 1)$ is an invariant of the program, which is a strong evidence that the implementation is good.

```
% x'*P*x<=1;
while (t<5000)
% x'*P*x<=1, wp(u=C*x+D*y,wp(x=A*x+B*y,x'*P*x<=1))=(A*x+B*y)'*P*(
    A*x+B*y)<=1
  u=C*x+D*y;
% wp(x=A*x+B*y,x'*P*x<=1)=(A*x+B*y)'*P*(A*x+B*y)<=1
  x=A*x+B*y;
% x'*P*x<=1
end
% x'*P*x<=1;
```

Fig. 5.17 Annotated lead/lag compensator in matlab

5.6.3 Strongest Post-condition

The dual of weakest pre-condition is the *strongest post-condition*. The strongest post-condition of C is a function that maps any pre-condition P to a post-condition. The output of the strongest post-condition function $sp(C, P)$ is the smallest set that holds after the execution of C, given that P holds before the execution of C. To verify a Hoare triple $\{P\}\, C\, \{Q\}$ using sp-calculus, first compute $sp(C, P)$ and then check that $sp(C, P) \to Q$.

5.7 Translation Process for a Simple Dynamical System

This section describes the credible autocoding process for a simple dynamical system, using a mixture of mathematics, C and ACSL.

The process starts with computing a quadratic invariant set for the system. Given a dynamical system $\mathscr{G}$ defined by $x_{+} = Ax$, the ellipsoid set $p(P, x, 1)$, constructed by solving $A^{\mathsf{T}}PA - P \prec 0$ for $P \succ 0$, is also invariant w.r.t to $\mathscr{G}$. The invariant property of $p(P, x, 1)$ is the key that allows us to know a priori that the Hoare Triple $\{p(P, x, 1)\}\, \mathscr{P}_2\, \{p(P, x, 1)\}$, in which $\mathscr{P}_2$ is a code implementation of $\mathscr{G}$ in Fig. 5.18, is correct. Since P is invertible, then $q(Q, x, 1)$ with $Q = P^{-1}$ is equivalent to $p(P, x, 1)$. The credible autocoder inserts $q(Q, x, 1)$ as the pre and post-conditions of the program.

Using the weakest pre-condition function from (5.20) on $q(Q, x, 1)$, one obtains $q(Q, x, 1)$ as the pre and post-conditions of the loop body in $\mathscr{P}_2$. Note that the set $q(Q, x, 1)$ is inserted into the code as pre and post-condition of the loop body. This is displayed in lines 7 and 8 of Fig. 5.18, with the loop body enclosed in curly braces.

```
 1 /*@
 2   requires q(Q,x,1);
 3   ensures  q(Q,x,1);
 4 */
 5 while (true) {
 6 /*@
 7   requires q(Q,x,1);
 8   ensures  q(Q,x,1);
 9 */
10 {
11   y1=0.4990*x1+0.1*x2;
12   y2=0.01*x1+1.0*x2;
13   x1=y1;
14   x2=y2;
15   }
16 }
```

Fig. 5.18 $\mathscr{P}_2$: code implementation of $\mathscr{G}$

Next, given the pre-condition $q(Q, x, 1)$ on the loop body, the strongest post-condition computations i.e. *sp*-calculus is performed on the code. Denote the body of the while loop in $\mathscr{P}_2$ as B, the credible autocoding process computes $sp(B, q(Q, x, 1))$ and then checks that $sp(B, q(Q, x, 1)) \to q(Q, x, 1)$ to ensures the correctness of $\{q(Q, x, 1)\}\, B\, \{q(Q, x, 1)\}$.

The *sp*-calculus process uses ellipsoidal calculus. One of the techniques from ellipsoidal calculus is the following regarding linear transformation of ellipsoidal sets.

Lemma 5.1 *Given a set $q(Q, x, 1)$, and given a linear transformation T, the image $T(q(Q, x, 1))$ is the set $q(TQT^{\mathsf{T}}, x, 1)$.*

Using the formula TQT^{T}, we can compute a strongest post-condition for every line of code in B. Define C_i as the ith line of code in B. Denote x_i as the state vector after the execution of C_i. For example, the state vector starts with $x = \begin{bmatrix} x1 \\ x2 \end{bmatrix}$ before the execution of C_1. The 2 lines of code C_1 and C_2 respectively assigns some values to the variable $y1$ and $y2$. The state vector's dimension increases and becomes $x_2 = \begin{bmatrix} x1 \\ x2 \\ y1 \\ y2 \end{bmatrix}$ after the execution of C_2. The state vector is x again after the execution of C_4. Because the variables $y1$ and $y2$ are discarded from the state vector when they are not used in the code again. Next, given state vectors x_{i-1} and x_i, and given the line of code C_i, the affine semantics of C_i is computed and then used in the construction of a linear transformation T_i from x_{i-1} to x_i. For example, for C_1, the code computes the expression $0.4990 * x1 + 0.1 * x2$ and assigns it to the variable $y2$. The affine semantics of C_1 is therefore $y1 = Lx$, in which $L = \begin{bmatrix} 0.4990 & 0.1 \end{bmatrix}$. The state vector x_0 is x and the state vector x_1 is $x_1 = \begin{bmatrix} x1 \\ x2 \\ y1 \end{bmatrix}$. Hence $T_1 = \begin{bmatrix} I \\ L \end{bmatrix}$. Applying Lemma 5.1, the strongest post-condition for C_m is

$$q(\prod_{i=m}^{1} T_i Q \prod_{i=1}^{m} T_i^{\mathsf{T}}, x_m, 1). \tag{5.22}$$

Hence the strongest-post condition for B i.e. $sp(B, q(Q, x, 1))$ is $q(Q_4, x, 1)$, in which

$$Q_4 = T_4 T_3 T_2 T_1 Q T_1^{\mathsf{T}} T_2^{\mathsf{T}} T_3^{\mathsf{T}} T_4^{\mathsf{T}} \tag{5.23}$$

The computed post-conditions are inserted into $\mathscr{P}_2$ (see Fig. 5.19) as the necessary evidence for the proof-checking of $\mathscr{P}_2$. To verify that $sp(B, q(Q, x, 1)) \implies q(Q, x, 1)$, the inclusion condition $q(Q_4, x, 1) \subseteq q(Q, x, 1)$ is checked. This can be done using a Cholesky decomposition algorithm to check that $Q - Q_4 \succ 0$.

```
while (1) {
/*@
  requires q(Q,x,1);
  ensures  q(T1*Q*T1',x1,1);
*/
{
  y1=0.4990*x1+0.1*x2;
}
/*@
  requires q(Q,x,1);
  ensures  q(T2*T1*Q*T1'*T2',x2,1);
*/
{
  y2=0.01*x1+1.0*x2;
}
/*@
  requires q(Q,x,1);
  ensures  q(T3*T2*T1*Q*T1'*T2'*T3',x3,1);
*/
{
  x1=y1;
}
/*@
  requires q(Q,x,1);
  ensures  q(T4*T3*T2*T1*Q*T1'*T2'*T3'*T4',x,1);
*/
{
  x2=y2;
}
  }
```

Fig. 5.19 $\mathscr{P}_2$ annotated

5.8 Gene-Auto+: A Prototype Credible Autocoder

In this section, some details of the prototype credible autocoder are given. The current prototype is capable of translating control semantics, described in Sect. 5.3.3 into verifiable ACSL annotations on the code.

5.8.1 Gene-Auto: Translation

Gene-Auto's translation architecture is comprised of sequences of independent model transformation stages. This classical, modular approach to code generator design has the advantage of allowing relatively easy insertion of additional transformation and formal analysis stages, such as the annotation generation stage in the prototype. The translation process goes through two layers of intermediate languages. The first one, called the *GASystemModel*, is a data-flow language that is similar to Simulink.

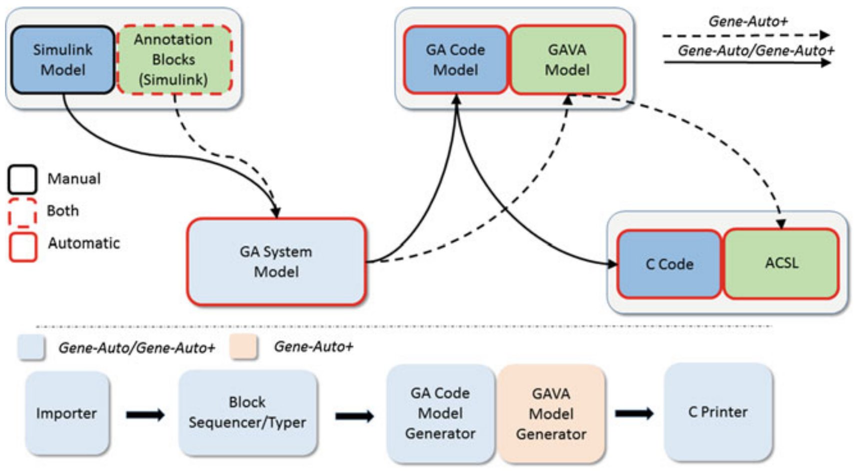

Fig. 5.20 Translation in gene-auto+ versus gene-auto

The input Simulink model, after being imported, is first transformed into the system model. The system model, which is expressed in the *GASystemModel* language, is then transformed into the code model. The code model is in the *GACodeModel* language representation, which has many similarities with imperative programming languages, such as C or Ada. The main translation modules within Gene-Auto, are the importer, the block sequencer and typer, the *GACodeModel* generator, and the C printer. For the prototype, we have recycled much of the transformation modules up to the *GACodeModel* generator. For the translation of the control semantics, we added a sub-module, dubbed the *GAVAModel* generator, to the *GACodeModel* generator. The *GAVAModel* is the ACSL-like language extension in Gene-Auto+. For more details about it, including its meta-model, please see [30]. The *GAVAModel* language enables common ASCL constructs such as: *behavior*, *assumes*-statement, *function contract*, *require*-statement, *ensure*-statement, and *ghost code* to be expressed within an intermediate representation in Gene-Auto+.

Figure 5.20 summarizes the key differences between the translation process of Gene-Auto and Gene-Auto+. The upper half of the figure shows the process in terms of languages and intermediate representations while the bottom part of the figure shows the translation modules. Of the four language representations in the translation process, only the *GASystemModel* representation remains unchanged. This is because, structurally speaking, the annotation blocks are identical to the non-annotation blocks.

5.8.2 *Translation of Annotative Blocks*

The annotation blocks are also first transformed into a *GASystemModel* representation. This transformation step is unchanged from the original Gene-Auto as the

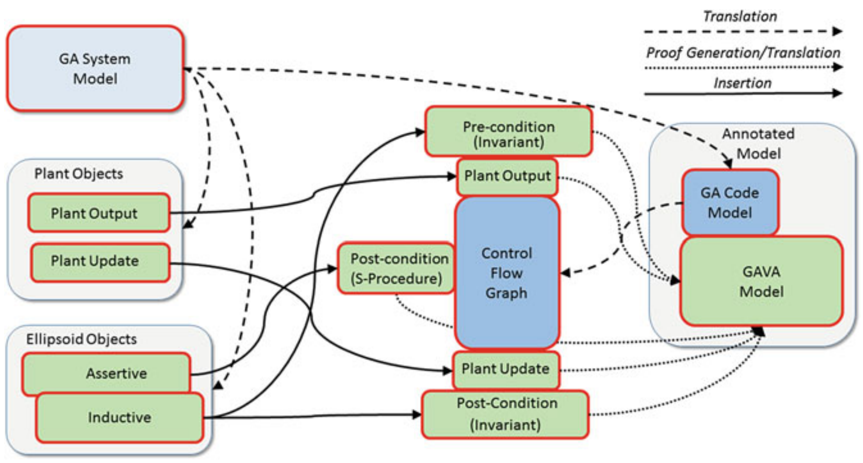

Fig. 5.21 Transformation of control semantics from *GASystemModel* to *GAVAModel*

same language is used to express both regular blocks and annotation blocks. In the *GACodeModel* generation stage, the blocks that express the control semantics are skipped since they are categorized as annotations. They are imported into the *GAVAModel* generation sub-module. This sub-module first translates the annotative blocks into a set of Hoare triple objects on the code model, and then translates the Hoare triple objects into a *GAVAModel* representation. This new representation of the code model with axiomatic semantics is dubbed the *annotated model*.

A high-level overview of the *GAVAModel* generator sub-module is summarized in Fig. 5.21. Following Gene-Auto's modular transformation architecture, the *GAVAModel* sub-module is added as an independent stage within the *GACodeModel* generation module. The major stages in the translation of the annotation blocks into the annotated model are the following:

1. The code model is converted into a control-flow graph structure $\mathscr{X}$.
2. The *constant* blocks are inserted into $\mathscr{X}$.
3. Constant propagation is executed with the definitions provided by the *constant* blocks.
4. The system block is translated into two *plant* objects. A *plant* object is comprised of an affine transformations and a set of ghost code templates expressed in *GAVAModel*. The plant objects are inserted into the beginning of $\mathscr{X}$ and the end of $\mathscr{X}$. The first plant object corresponds to the output function of the state-space system $y = Cx$. The second plant object corresponds to the state-transition function $x_+ = Ax + Bu$.
5. The *quadratic* blocks are grouped based on their inputs as either inductive or assertive. They are translated into *ellipsoid* objects and inserted into appropriate locations within $\mathscr{X}$.

6. The strongest post-condition is computed using *sp*-calculus. In this process, *ellipsoid* objects are generated for almost every line of code and then inserted into the code model.
7. The ellipsoid and plant objects in the annotation model are translated into annotations expressed in *GAVAModel*.

5.9 Translation and Insertion of the *System Block*

The *system* block, which represents the model of the plant, is split into two plant objects representing two linear transformations: $y = Cx$ and $x_+ = Ax + Bu$. One object is inserted into the beginning of the *compute* function and the other part is inserted afterwards. The *compute* function is the function that implements the controller loop body. The two linear transformations are used in the *sp*-calculus to be described later. The *GAVAModel* templates, contained within the two plant objects, are translated into a set of ACSL ghost code statements. The set of ACSL ghost code statements, generated from the closed-loop example, is displayed in Fig. 5.22.

```
/*@
        requires _io_->y == 1.0 * Plant_0_1[0];
*/
void cl_result_compute(t_cl_result_io *_io_, t_cl_result_state *
    _state_) {
.
.
.
}
 {
    /*@
            ghost Plant_xp_0_tmp = Plant_0_1[0];
    */
    /*@
            ghost Plant_xp_1_tmp = Plant_0_1[1];
    */
    /*@
            ghost Plant_0_1[0] = 1.0 * Plant_xp_0_tmp + 0.01 *
                Plant_xp_1_tmp + 5.0E-5 * _io_->u;
    */
    /*@
            ghost Plant_0_1[1] = -0.01 * Plant_xp_0_tmp + 1.0 *
                Plant_xp_1_tmp + 0.01 * _io_->u;
    */

 }
```

Fig. 5.22 Ghost code representation of the plant dynamics

5.10 Translation of the *Quadratic* Blocks

A short description of the typing of the *quadratic* blocks and their translations is given here. The semantics of closed-loop stability are structured in such way that there is one inductive ellipsoid set (the Lyapunov function), on the model with another ellipsoid set on the input (bounded input).

5.10.1 Types of Quadratic Blocks

The *quadratic* blocks are separated into two main groups. The first group include the blocks that are inductive. These encode the stability property of the system. To determine if a *quadratic* block is inductive, the following properties must be computed:

1. Every one of the *quadratic* block's input ports must be connected to a port of an unit delay block or to an output port of a *system* block.
2. Given a set $\mathscr{U}$ that contains all unit delay blocks connected to the *quadratic* block. For every unit delay blocks in $\mathscr{U}$, there exists a path from its output node to its input node on the system model.

The second group contains the assertive blocks. These blocks are used to either express boundedness of inputs or sector-bound conditions. Any *quadratic* blocks with one or more input connected to a block that is neither unit delay nor *system* is categorized as an assertive block. The sector-bound blocks are detected by checking to see if the level-set parameter c in the block is set to 0.

After the *quadratic* blocks have been categorized, the bounded-input and inductive blocks are translated into ellipsoid objects containing the Schur form of $p(P, x, 1)$ i.e. $q(Q, x, 1)$ such that $Q = P^{-1}$. This conversion is necessary as all subsequent *sp*-calculus are done in the Schur form due to the possibility of Q_i in $q(Q_i, x_i, 1)$ being singular.

5.10.2 Insertion of Ellipsoid Objects

An assertive ellipsoid invariant $q(Q, x, 1)$ is inserted into a location that is dependent on x. If any of the variable in x is an input argument of the `compute` function, then algorithm will back propagate using *wp*-calculus until x only contains either variables that are input arguments of the compute function, or affine expressions of the variables that are input arguments of the `compute` function.

The *wp*-calculus starts, if needed, after the assertive ellipsoid $q(Q, x, 1)$ has been inserted as a post-condition for the last line of code, in which, a variable belonging to x is assigned. For example, consider the annotation block *bounded_input* in the open-loop case, which expresses a boundedness assumption on the signal $y - y_d$.

```
 1 /*@
 2     requires in_ellipsoidQ(QMat_0,vect_of_1_scalar(_io_->y - _io_
           ->input));
 3 */
 4 void simple_olg_compute(t_simple_olg_io *_io_, t_simple_olg_state
        *_state_) \{
 5 .
 6 .
 7 .
 8     /*@
 9        requires in_ellipsoidQ(QMat_0,vect_of_1_scalar(_io_->y -
              _io_->input));
10        ensures in_ellipsoidQ(QMat_0,vect_of_1_scalar(_io_->y -
              simple_olg_input));
11     */
12  {
13     simple_olg_input = _io_->input;
14  }
15
16     /*@
17
18         requires in_ellipsoidQ(QMat_0,vect_of_1_scalar(_io_->y -
               simple_olg_input));
19   ensures in_ellipsoidQ(QMat_4,vect_of_1_scalar(_io_->y -
         simple_olg_input));
20
21     */
22 .
23 .
24 .
25    /*@
26       requires in_ellipsoidQ(QMat_14,vect_of_1_scalar(
             simple_olg_y - simple_olg_input));
27       ensures in_ellipsoidQ(QMat_14,vect_of_1_scalar(Sum4));
28     */
29 {
30     Sum4 = simple_olg_y - simple_olg_input;
31 }
```

Fig. 5.23 *wp*-calculus on an ellipsoids expressed in ACSL

The signal $y - y_d$ also corresponds to the variable **Sum4** in Fig. 5.23. The annotation block is translated into an assertive ellipsoid object and is inserted into the code as a post-condition of `Sum4=simple_olg_y-simple_olg_y_input`. This post-condition is displayed in the last ACSL contract of Fig. 5.23. Since the variable `Sum4` is not an argument of the `compute` function, the insertion algorithm starts the *wp*-calculus, until x_{-n} in $Q(Q, x_{-n}, 1)$ only contains variables that are input arguments of the `compute` function. For this case, the *wp*-calculus terminated when the ellipsoid in line 2 of Fig. 5.23 is generated.

The insertion of an inductive ellipsoid is more straightforward. The inductive ellipsoid is duplicated three times and inserted as pre and post-conditions respectively at the beginning and end of the `compute` function body. It is also inserted as a pre

```
/*@
   requires in_ellipsoidQ(QMat_1,vect_of_2_scalar(_state_->
       Integrator_1_memory,_
              state_->Integrator_2_memory));
   ensures in_ellipsoidQ(QMat_1,vect_of_2_scalar(_state_->
       Integrator_1_memory,_
              state_->Integrator_2_memory));
*/
void simple_olg_compute(t_simple_olg_io *_io_, t_simple_olg_state
     *_state_) {
.
.
.
 /*@
   requires in_ellipsoidQ(QMat_1,vect_of_2_scalar(_state_->
       Integrator_1_memory,_
              state_->Integrator_2_memory));
   ensures in_ellipsoidQ(QMat_2,vect_of_2_scalar(_state_->
       Integrator_1_memory,_
              state_->Integrator_2_memory));
 */
 {
    simple_olg_input = _io_->input;
 }
 .
 .
 .
 /*@
   requires in_ellipsoidQ(QMat_29,vect_of_2_scalar(_state_->
       Integrator_1_memory,
              _state_->Integrator_2_memory));
   ensures in_ellipsoidQ(QMat_1,vect_of_2_scalar(_state_->
       Integrator_1_memory,
          _state_->Integrator_2_memory));
 */
  {

  }
 }
 // end of the function
```

Fig. 5.24 Inductive ellipsoids in ACSL

and post-conditions on the function itself. These ellipsoids are the ones defined by the matrix variable `QMat_1` in Fig. 5.24.

5.11 Computing the Strongest Post-condition

The *sp*-calculus has been automated in Gene-Auto+ using a set of transformation rules from ellipsoidal calculus, which are used to compute $sp(q(Q, x, 1), C)$ or its over-approximation for various block of code C. The set of transformation rules can be divided into two categories: affine transformations, and S-Procedure transformations.

5.11.1 Affine Transformation

The basics of affine transformation have been described using the example $x_+ = Ax$ in Sect. 5.7 and proven in PVS (see Fig. 5.11). For automating the proof-checking of the affine transformations of ellipsoids, we define a proof tactic denoted *AffineEllipsoid*, which corresponds to a proof strategy of the same name defined in PVS. This rule is applied whenever a linear abstraction of the code can be computed. Recall from Sect. 5.7, given the pre-condition $q(Q_i, x_i, 1)$ and the code $z = Lx_i$, then the linear transformation from x_i to x_{i+1} is $T_i = \begin{bmatrix} I \\ L \end{bmatrix}$. The strongest post-condition is therefore $q(T_i Q_i T_i^{\mathrm{T}}, x_{i+1}, 1)$.

In the more general case, let the affine semantic of a block of code be $z := Ly$, where $y \in \mathbb{R}^m$ is vector of program states and $L \in \mathbb{R}^{1\times m}$. Let $\mathscr{Q}_i(x) := q(Q_i, x, 1)$, then the *AffineEllipsoid* tactic is

$$\frac{}{\{\mathscr{Q}_n(x)\}\, z := a\, \{\mathscr{Q}_{n+1}(x \cup z)\}}, Q_{n+1} = \mathscr{F}\left(Q_n, \psi(L, y, x), \phi(z, x)\right), \tag{5.24}$$

and the function $\mathscr{F}$ is defined as follows: given the functions $\psi : (L, y, x) \to \mathbb{R}^{1\times n}$ and $\phi : (z, x) \to \mathbb{Z}$, we have

$$\begin{gathered}
\mathscr{F} : (Q_n, \psi(L, y, x), \phi(z, x)) \to T\left(\psi(L, y, x), \phi(z, x)\right)^{\mathrm{T}} Q_n T\left(\psi(L, y, x), \phi(z, x)\right) \\
T\left(\psi(L, y, x), \phi(z, x)_{i,j}\right) := \begin{cases} 1, & 0 \le i, j \le n \wedge i = j \wedge i \ne \phi(z, x) \\ 0, & 0 \le i, j \le n \wedge i \ne j \wedge i \ne \phi(z, x) \\ \psi(y, x)_{1,j}, & i = \phi(z, x) \wedge 0 \le j \le n \end{cases} \\
\psi(L, y, x)_{1,j} := \begin{cases} L(1, k), & 0 \le j, k \le n \wedge x_j \in y \wedge y_k = x_j \\ 0, & 0 \le j \le n \wedge x_j \notin y \end{cases} \\
\phi(z, x) := \begin{cases} i, & z \in x \wedge z = x_i \\ n+1, & z \notin x \end{cases}
\end{gathered} \tag{5.25}$$

The *ReduceEllipsoid* tactic is used, when the state x_i of the program is reduced in dimensions from the previous state x_{i-1}. Let $\mathscr{Q}(x) := q(Q, x, 1)$, then the *ReduceEllipsoid* tactic is

$$\frac{}{\{\mathscr{Q}_n(x)\}\, \mathbf{SKIP}\, \{\mathscr{Q}_{n+1}(x \setminus \{z\})\}}, Q_{n+1} = \mathscr{G}\left(Q_n, \theta(z, x)\right), \tag{5.26}$$

and the function $\mathscr{G}$ is defined as the following: given the function $\theta : (z, x) \to \mathbb{Z}$, we have

$$\begin{gathered}\mathscr{G} : (Q_n, \theta(z,x)) \to T\left(\theta(z,x)\right)^{\mathrm{T}} Q_n T\left(\theta(z,x)\right) \\ T\left(\theta(z,x)_{i,j}\right) := \begin{cases} 1, \ 0 \le i,j \le n-1 \wedge ((i < \theta(z,x) \wedge i = j) \vee (i \ge \theta(z,x) \wedge j = i+1)) \\ 0, \ 0 \le i,j \le n-1 \wedge ((i < \theta(z,x) \wedge i \ne j) \vee (i \ge \theta(z,x) \wedge j \ne i+1)) \end{cases} \\ \theta(z,x) := \left\{ i, \ z = x_i \right.\end{gathered} \tag{5.27}$$

Before the insertion of the Ellipsoid objects into the code model, each line of code is analyzed for its affine semantics. A linear transformation matrix L is extracted from the abstract semantics and stored in the control flow graph. For example, if we have $x = y + 2z$, then the affine algorithm returns a linear function represented by the matrix $L = \begin{bmatrix}1 \ 2\end{bmatrix}$. For the plant objects, their affine semantics are computed from the templates stored in the semantics of the *system* blocks.

Figure 5.25 shows an example of the *AffineEllipsoid* usage in the open-loop example. In this example, the pre-condition is the ellipsoid defined by the matrix variable `QMat_21`, and the ensuing line of code assigns the expression `dt_+x1` to the variable `Sum2`. The affine transformation matrix is $L = \begin{bmatrix}1 \ 1\end{bmatrix}$, and by applying the *AffineEllipsoid* rule, the ellipsoid transformation matrix T is

$$T = \begin{cases} T_{ij} = 1.0, & (i \le 4 \wedge i = j) \vee (i = 6 \wedge (j = 6 \vee i = 6)) \vee (i = 5 \wedge j = 6) \\ T_{ij} = 0.0, & \text{otherwise.} \end{cases} \tag{5.28}$$

5.11.2 S-Procedure

The *S-Procedure* tactic, proven in PVS (see Fig. 5.12), is used to compute an over-approximation of the strongest post-condition for the nonlinear portion of the code. It is based on the well-known principle of Lagrangian relaxation for quadratic forms [31]. For the bounded input stability problem, the LMI solution also yields a small positive multiplier $1 >> \lambda > 0$ that is associated with the bounded input quadratic form. This small multiplier proves to be useful in the *sp*-calculus in the following sense. Consider the line of code `yc=yd-z` with two pre-conditions. One of the two pre-conditions is $q(Q_b, yd, 1)$, which is translated from the *quadratic* block *bounded_input* in the closed-loop model. The other pre-condition $q(Q_i, x_i, 1)$ $y \in x_i$, is generated from the *sp*-calculus. The multiplier $\lambda > 0$ enables us to combine in a convex fashion the two pre-conditions $q(Q_b, yd, 1)$ and $(Q_i, x_i, 1)$ into a single post-condition $q(Q_{i+1}, x_{i+1}, 1)$, $x_{i+1} = x_i \cup \{yd\}$, in which

$$Q_{i+1} = \begin{bmatrix} \frac{1}{1-\lambda} Q_i & 0 \\ 0 & \frac{1}{\lambda} Q_b \end{bmatrix} \tag{5.29}$$

```
/*@
        logic matrix QMat_21 = mat_mult(mat_mult(
  mat_of_6x7_scalar
      (1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,
        0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,
        1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,
        0.0,0.0,0.0,0.0,0.0,0.01),QMat_20),
        transpose(mat_of_6x7_scalar(1.0,0.0,0.0,0.0,0.0,0.0,
        0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,
        0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,
        0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.01)));
*/
/*@
        logic matrix QMat_22 = mat_mult(mat_mult(
  mat_of_6x6_scalar
      (1.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,
        0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,
        0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,1.0,0.0,0.0,1.0),QMat_21)
            ,
    transpose(
  mat_of_6x6_scalar
      (1.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,
        0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,
        0.0,0.0,0.0,0.0,0.0,1.0, 0.0,0.0,1.0,0.0,0.0,1.0)));
*/
    /*@
            behavior ellipsoid17_0:
            requires in_ellipsoidQ(QMat_21,
      vect_of_6_scalar(_state_->Integrator_1_memory,
      _state_->Integrator_2_memory,x1,Sum3,Sum1,dt_));
            ensures in_ellipsoidQ(QMat_22,
      vect_of_6_scalar(_state_->Integrator_1_memory,
      _state_->Integrator_2_memory,x1,Sum3,dt_,Sum2));
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
    */
{
    Sum2 = dt_ + x1;
}
```

Fig. 5.25 Application of *AffineEllipsoid*

Let $\mathscr{Q}_n(x_i) := q(Q_n, x_i, 1)$, the *S-Procedure* tactic is

$$\frac{}{\{\mathscr{Q}_1(x_1) \wedge \mathscr{Q}_2(x_2) \wedge \ldots \wedge \mathscr{Q}_N(x_N)\}\ \mathbf{SKIP}\ \{\mathscr{Q}_{n+1}(x_0 \cup x_1 \cup \ldots \cup x_n)\}} \qquad Q_{n+1} = \sum_{i=1}^{N} \mu_i \mathscr{H}(Q_i)\,, \tag{5.30}$$

and $\mathscr{H} : \mathbb{R}^{n_i \times n_i} \to \mathbb{R}^{Nn \times Nn}$ is defined as follows: given the function $\dim : R^{n \times n} \to n$, and the function $\rho : n \in \mathbb{Z}^+ \to \sum_{i=1}^{n} \dim(Q_i)$, we have

$$\mathscr{H}(Q_i)(n,m) = \begin{cases} Q_i(n-\rho(i-1), m-\rho(i-1)), & \rho(i-1) \le n, m \le \rho(i) \\ 0.0, & \text{otherwise} \end{cases} \tag{5.31}$$

Given the pre-condition $\{\mathscr{Q}_i(x_i)\}$ and code C such that $[\![C]\!] \vDash (y := Lz)$, the *SProcedure* rule is activated only when all the following conditions are satisfied:

1. For each $\mathscr{Q}_i(x_i)$, the *AffineEllipsoid* rule does not apply.
2. For the set $\{\mathscr{Q}_i(x_i)\}, i = 1, \ldots, N, z \subseteq \bigcup_{i=1}^{N} x_i$.
3. For $\mathscr{Q}_i(x_i), i = 1, \ldots, N, z \nsubseteq x_i \wedge z \cap x_i \neq \{\varnothing\}$.

The multipliers are computed beforehand using the S-Procedure theory to ensure that the *sp*-calculus, which uses the *S-Procedure* rule at some point, does not result in a strongest post-condition that violates the initial pre-condition. For the closed-loop example in Fig. 5.26, there is one ellipsoid pre-condition defined by the matrix variable `QMat_12`. This ellipsoid is translated from the *quadratic* block *bounded_input*. The other ellipsoid pre-condition, is defined by the matrix variable `QMat_13`. This ellipsoid is computed by the *sp*-calculus. These two ellipsoids are combined to form a post-condition ellipsoid using the *S-Procedure*. The matrix variable `QMat_14`, which defines the post-condition ellipsoid, is expressed using the pre-defined ACSL block matrices function `block_m`.

```
/*@
        logic matrix QMat_14 = block_m(
  mat_scalar_mult(1.0009008107296566,QMat_13),
  zeros(6,2),
  zeros(2,6),
  mat_scalar_mult(1111.111111111111,QMat_12));
*/
    /*@
            behavior ellipsoid9_0:
            requires in_ellipsoidQ(QMat_13,
      vect_of_6_scalar(_state_->Integrator_1_memory,
      _state_->Integrator_2_memory,x1,C11,Integrator_2,Sum3));
            requires in_ellipsoidQ(QMat_12,vect_of_2_scalar(Sum4,
                D11));
            ensures in_ellipsoidQ(QMat_14,
      vect_of_8_scalar(_state_->Integrator_1_memory,
      _state_->Integrator_2_memory,x1,C11,Integrator_2,Sum3,Sum4,
          D11));
            @ PROOF_TACTIC (use_strategy (SProcedure));
    */
{

}
```

Fig. 5.26 Application of the *S-Procedure* tactic

```
/*@
     behavior Plant_22:
          requires in_ellipsoidQ(QMat_32,vect_of_4_scalar(
             _state_->Integrator_1_memory,_state_->
             Integrator_2_memory,Plant_0_1[0],Plant_0_1[1]));
          ensures in_ellipsoidQ(QMat_1,vect_of_4_scalar(_state_
             ->Integrator_1_memory,_state_->
             Integrator_2_memory,Plant_0_1[0],Plant_0_1[1]));
          @ PROOF_TACTIC (use_strategy (PosDef));
   */
{

}
```

Fig. 5.27 Verifying the strongest post-condition

5.11.3 Verification of the Strongest Post-condition

After application of the *sp*-calculus, the alternative post-condition generated must be checked against the initial pre-condition. For the closed-loop example displayed in Fig. 5.27, this means checking if `Q_1` is a "bigger" matrix than `Q_32`. If this verification condition can be discharged, then one can claim the code satisfies the stability property. For the closed-loop example, because of a subtle error introduced into the model that went unnoticed until this point, the last verification condition could not be discharged until the sign error was corrected in the Simulink model.

5.12 Automatic Verification of Control Semantics

As part of the credible autocoding process, the annotated C code which generation process we described in Sect. 5.4, must be independently verifiable. Indeed, we now describe and implement a tool that can be used by the certification authority in order automatically check that the annotations are correct with respect to the code. This is achieved by checking that each of the individual Hoare triples hold. Figure 5.28 presents an overview of the checking process. First, the WP plugin of Frama-C generates verification conditions for each Hoare triple, and discharges the trivial ones with its internal prover QeD. Then, the remaining conditions are translated into PVS theorems for further processing, as described in Sect. 5.12.1. It is then necessary to match the types and predicates introduced in ACSL to their equivalent representation in PVS. This is done through theory interpretation [32] and explained in Sect. 5.12.2. Once interpreted, the theorems can be generically proven thanks to custom PVS strategies, as described in Sect. 5.12.3. In order to automatize these various tasks and integrate our framework within the Frama-C platform, which provides graphical support to display the status of a verification condition (proved/unproved), we wrote a Frama-C plugin named pvs-ellipsoid, described in Sect. 5.12.4. Finally, one verification condition does not fall under either `AffineEllipsoid` of `SProcedure` strategies. It is discussed in Sect. 5.12.5.

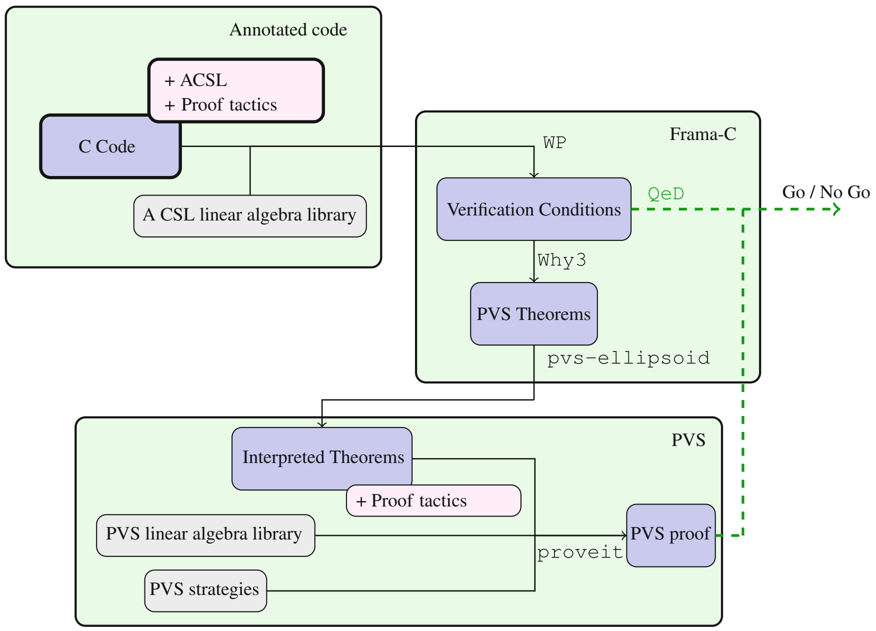

Fig. 5.28 General view of the automated verification process described and implemented in this section

5.12.1 From C Code to PVS Theorems

The autocoder described in the previous Section generates two C functions. One of them is an initialization function, the other implements one execution of the loop that acquires inputs and updates the state variables and the outputs. It is left to the implementer to write the main function combining the two, putting the latter into a loop, and interfacing with sensors and actuators to provide inputs and deliver outputs. Nevertheless, the properties of open and closed loop stability, as well as state-boundedness, can be established by solely considering the update function, which this section now focuses on. The generated function essentially follows the template shown in Fig. 5.29.

Frama-C is a collaborative platform designed to analyze the source code of software written in C. The WP plugin enables deductive verification of C programs annotated with ACSL. For each Hoare tripe $\{pre_i\}\text{inst}_i\{post_i\}$, it generates a first order logic formula expressing $pre_i \implies wp(\text{inst}_i, post_i)$.[2] Through the Why3 platform, these formulas can be expressed as theorems in PVS, so that, for example, the ACSL/C triple shown in Fig. 5.30, taken directly from our running example, becomes the theorem shown in Fig. 5.31.

[2]Given a program statement S and a postcondition Q, $wp(S, Q)$ is the weakest precondition on the initial state ensuring that execution of S terminates in a state satisfying Q.

```
/*@ requires in_ellipsoidQ(Q,
                           vect_of_n_scalar(_state_->s_1,
                                            _state_->s_2,
                                            ...));
  @ ensures in_ellipsoidQ(Q,vect_of_n_scalar(_state_->s_1,
                                            _state_->s_2,
                                            ...));*/
void example_compute(t_example_io *_io_,
                     t_example_state *_state_){
...
/*@ requires pre_i
  @ ensures post_i
  @ PROOF_TACTIC (use_strategy ( strategy_i ) )*/
{
instruction i;
}
...
}
```

Fig. 5.29 Template of the generated loop update function

```
/*@
requires in_ellipsoidQ(QMat_4,
                       vect_of_3_scalar(_state_->
                           Integrator_1_memory,
                                    _state_->
                                        Integrator_2_memory,
                                    Integrator_1));
ensures in_ellipsoidQ(QMat_5,
                      vect_of_4_scalar(_state_->
                          Integrator_1_memory,
                                   _state_->
                                       Integrator_2_memory,
                                   Integrator_1,
                                   C11));
PROOF_TACTIC (use_strategy (AffineEllipsoid));
*/
{
    C11 = 564.48 * Integrator_1;
}
```

Fig. 5.30 Typical example of an ACSL Hoare triple

Note that, for the sake of readability, part of the hypotheses of this theorem, including hypotheses on the nature of variables, as well as hypotheses stemming from Hoare triples present earlier in the code, are ommitted here. Note also that in the translation process, functions like `malloc_0` or `mflt_1` have appeared. They describe the memory state of the program at different execution points.

PVS
```
wp: THEOREM
FORALL (integrator_1_0: real):
  FORALL (malloc_0: [int -> int]):
    FORALL (mflt_2: [addr -> real], mflt_1: [addr -> real],
            mflt_0: [addr -> real]):
     FORALL (io_2: addr, io_1: addr, io_0: addr, state_2: addr,
             state_1: addr, state_0: addr):
             ...
     => p_in_ellipsoidq(l_qmat_4,
                        l_vect_of_3_scalar(mflt_2(shift
                                                   (state_2, 0)),
                                           mflt_2(shift
                                                   (state_2, 1)),
                                           integrator_1_0))
     => p_in_ellipsoidq(l_qmat_5,
                        l_vect_of_4_scalar(mflt_2(shift
                                                   (state_2, 0)),
                                           mflt_2(shift
                                                   (state_2, 1)),
                                           integrator_1_0,
                                           (14112/25 *
                                             integrator_1_0)))
```

Fig. 5.31 Excerpt of the PVS translation of the triple shown in Fig. 5.30

5.12.2 *Theory Interpretation*

At the ACSL level, a minimal set of linear algebra symbols has been introduced, along with axioms defining their semantics. Section 5.3 describes a few of them. Each generated PVS theorem is written within a theory that contains a translation 'as is' of these definitions and axioms, along with some constructs specific to handling the semantics of C programs. For example, the ACSL axiom expressing the number of rows of a 2 by 2 matrix (in Fig. 5.32) becomes the axiom shown in Fig. 5.33 after translation to PVS.

In order to leverage the existing results on matrices and ellipsoids in PVS, theory interpretation is used. It is a logical technique used to relate one axiomatic theory to another. It is used here to map types introduced in ACSL, such as vectors and matrices, to their counterparts in PVS, as well as the operations and predicates on these types. To ensure soundness, PVS requires that what was written as axioms in the ACSL library be proven in the interpreted PVS formalism.

```
/*@ axiom mat_of_2x2_scalar_row:
  @ \forall matrix A, real x0101, x0102, x0201, x0202;
  @ A == mat_of_2x2_scalar(x0101, x0102, x0201, x0202) ==>
  @ mat_row(A) == 2; /*
```

Fig. 5.32 ACSL axiomatization of 2 by 2 matrix row-size

```
PVS
q_mat_of_2x2_scalar_row:
 AXIOM FORALL (x0101_0:real, x0102_0:real,
              x0201_0:real, x0202_0:real):
           FORALL (a_0:a_matrix):
    (a_0 = l_mat_of_2x2_scalar(x0101_0, x0102_0,
                              x0201_0, x0202_0)) =>
    (2 = l_mat_row(a_0))
```

Fig. 5.33 Translation of the ACSL axiom from Fig. 5.32 into PVS

The interpreted symbols and soundness checks are the same for each proof objective, facilitating the mechanization of the process. Syntactically, a new theory is created, in which the theory interpretation is carried out, and the theorem to be proven is automatically rewritten by PVS in terms of its own linear algebra symbols. These manipulations on the generated PVS code are carried out by a frama-C plugin called pvs-ellipsoid, which is described below.

5.12.3 Generically Discharging the Proofs in PVS

Once the theorem is in its interpreted form, all that remains to do is to apply the proper lemma to the proper arguments. Section 5.4 describes two different types of Hoare triples that can be generated in ACSL. Two PVS strategies were written to handle these possible cases. A PVS proof strategy is a generic function describing a set of basic steps communicated to the interactive theorem prover in order to facilitate or even fully discharge the proof of a lemma. The `AffineEllipsoid` strategy handles any ellipsoid update stemming from a linear assignment of the variables. Recall `ellipsoid_general`, a theorem introduced in Sect. 5.3:

```
PVS
ellipsoid_general: LEMMA
 ∀ (n:posnat,m:posnat, Q:SquareMat(n),
           M: Mat(m,n), x:Vector[n], y:Vector[m]):
              in_ellipsoid_Q?(n,Q,x)
              AND y = M*x
         IMPLIES
         in_ellipsoid_Q?(m,M*Q*transpose(M),y)
```

To apply this theorem properly, the first step of the strategy consists of parsing the objectives and hypotheses of the theorem to acquire the name and the dimensions of the relevant variables, and to isolate the necessary hypotheses. The second step consists of a case splitting on the dimensions of the variable: they are given to the prover in order to complete the main proof, and then established separately using the proper interpreted axioms. Next, it is established that $y = Mx$ through a case decomposition and numerous calls to relevant interpreted axioms. All the hypotheses are then present for a direct application of the theorem. The difficulties in proof strategy design lie in intercepting and anticipating the typecheck constraints (tccs) that PVS introduces throughout the proof. A third strategy was specifically written to handle them.

The S-Procedure strategy follows a very similar pattern, somewhat simpler since the associated instruction in the Hoare triple is a `skip`, using `ellipsoid_combination`, the other main theorem presented in Sect. 5.3.

5.12.4 The `pvs-ellipsoid` Plugin to Frama-C

The pvs-ellipsoid plugin to Frama-C automatizes the steps mentionned in the previous subsections. It calls the WP plugin on the provided C file, then, whenever QeD fails to prove a step, it creates the PVS theorem for the verification condition through Why3 and modifies the generated code to apply theory interpretation. It extracts the proof tactic to be used on this specific verification condition, and uses it to signify to the next tool in the chain, `proveit` [33], what strategy to use to prove the theorem at hand. `proveit` is a command line tool that can be called on a PVS file and attempts to prove all the theories in it, possibly using user guidance such as the one just discussed. When the execution of `proveit` terminates, a report is produced, enabling the plugin to decide whether the verification condition is discharged or not. If it is, a proof file is generated, making it possible for the proof to be replayed in PVS.

5.12.5 Checking Inclusion of the Propagated Ellipsoid

One final verification condition falls under neither the `AffineEllipsoid` nor the `S-Procedure` categories. It expresses that the state remains in the initial ellipsoid $\mathscr{G}_P$. Through a number of transformations, we have proof that the state lies in some ellipsoid $\mathscr{G}'_P$. The conclusion of the verification lies in the final test $P - P' \geq 0$. The current state of the linear algebra library in PVS does not permit to make such a test, however a number of very reliable external tools, like the INTLAB package of the MATLAB software suite, can operate this check. In the case of our framework, the pvs-ellipsoid plugin intercepts this final verification condition before translating it to PVS, and uses custom code from [34] to ensure positive definiteness of the matrix, with the added benefit of soundness with respect to floating point computations.

5.13 Related Works

Although the efforts described in this chapter explore a new intersection between control theory and computer science, a few notable earlier works are mentioned here. Ursula Martin and her team developed a limited Hoare logic framework to reason on frequency domain properties of linear controllers at the Simulink level [35]. Jerome Ferret was the first to focus on the static analysis of digital filters in [5]. It was this work that initiated the connections made between the control-theoretic techniques and software analysis methods in [9]. Parallel work by Roux et al. [34] uses policy interation to generate and refine ellipsoid invariants. We would like to thank Eric Goubault and Sylvie Putot for the useful discussions, and mention their work on zonotopal domain for static analyzers [36].

5.14 Conclusion

The prototype tools and various examples described in this chapter can be found online.[3] We have demonstrated in this chapter a set of prototype tools that is capable of migrating high-level functional properties of control systems down to the code level. In addition we have developed a tool which can independantly verify the correctness of those properties for the code, in an automatic manner. While the nature of controllers and properties supported is relatively restricted, this effort demonstrates the feasability of a paradigm where domain specific knowledge is leveraged and automatically assists code analysis. This opens the way for numerous directions of research. As the mathematical breadth of theorem provers increase, increasingly complex code invariants can theoretically be handled, and thus increasingly complex controllers. In particular, generating verifiable optimization algorithms, e.g. for the purpose of path planning, is a promising direction. Soundness of the results with respect to floating point computations is another issue that requires attention.

The toolchain had been applied not only to the running example, but also on industry size systems, such as the Quanser 3 degree-of-freedom helicopter, and a very light jet turbofan engine controller from Price Induction.

Acknowledgments The authors would like to thank Pierre Roux for his contribution to the PVS-ellipsoid plugin, Gilberto Perez and Pablo Ascariz for their invaluable help on the PVS linear algebra library.

This chapter was prepared under support from NSF Grant CPS Medium 1135955 (CrAVES), CPS Synergy 1446758 (SORTIES), the Army Research Office under MURI W911NF-11-1-0046, the National Aeronautics and Space Administration under NASA Cooperative Agreement NNL09AA00A, activity 2736, the French Fond Unique Interministeriel 2011 project P, ITEA2 OPES, FNRAE project CAVALE, ANR INS project CAFEIN and ANR ASTRID project VORACE.

[3]http://www.feron.org/Eric/Credible.

References

1. Clarke Jr, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge, MA, USA (1999)
2. De Moura, L.: Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08, pp. 337–340. Springer, Berlin, Heidelberg (2008)
3. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Annual Design Automation Conference. DAC '01, pp. 530–535. ACM, New York, NY, USA (2001)
4. Souyris, J.: Industrial experience of abstract interpretation-based static analyzers. In: Jacquart, R. (ed.) Building the Information Society, IFIP International Federation for Information Processing, vol. 156, pp. 393–400. Springer, US (2004). doi:10.1007/978-1-4020-8157-6_31
5. Feret, J.: Static analysis of digital filters. In: European Symposium on Programming (ESOP'04), no. 2986 in LNCS, pp. 33–48. Springer, Berlin (2004)
6. Feret, J.: Numerical abstract domains for digital filters. In: International workshop on Numerical and Symbolic Abstract Domains (NSAD) (2005)
7. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE analyzer. In: Proceedings of the 14th European Symposium on Programming, Lecture Notes in Computer Science, vol. 3444 (2005)
8. Monniaux, D.: Compositional analysis of floating-point linear numerical filters. In: CAV (2005)
9. Feron, E.: From control systems to control software. IEEE Control Syst. **30**(6), 50–71 (2010)
10. Herencia-Zapana, H., Jobredeaux, R., Owre, S., Garoche, P.L., Feron, E., Perez, G., Ascariz, P.: Pvs linear algebra libraries for verification of control software algorithms in c/acsl. In: NASA Formal Methods, pp. 147–161 (2012)
11. Rinard, M.: Credible compilation. Techincal report, In: Proceedings of CC 2001: International Conference on Compiler Construction (1999)
12. Denney, E.: Certifying auto-generated flight code. http://shemesh.larc.nasa.gov/LFM2008/slides/Session3/Denney.ppt (2008)
13. Dieumegard, A.: Formal guarantees for safety critical code generation: the case of highly variable languages. Ph.D. thesis, INP Toulouse (2015)
14. Boyd, S., El Ghaoui, L., Feron, E., Balakrishnan, V.: Linear Matrix Inequalities in System and Control Theory, Studies in Applied Mathematics, vol. 15. SIAM, Philadelphia, PA (1994)
15. Megretski, A., Rantzer, A.: System analysis via integral quadratic constraints. IEEE Trans. Autom. Control **42**(6), 819–830 (1997)
16. Berry, G., Gonthier, G., Gonthier, A.B.G., Laltte, P.S.: The esterel synchronous programming language: Design, semantics, implementation (1992)
17. Pakmehr, M., Wang, T., Jobredeaux, R., Vivies, M., Feron, E.: Verifiable control system development for gas turbine engines. CoRR abs/1311.1885 (2013)
18. Yakubovich, V.A.: The solution of certain matrix inequalities in automatic control theory. Soviet Math. Dokl **3**, 620–623 (1962)
19. Nesterov, Y., Nemirovskii, A., Ye, Y.: Interior-point polynomial algorithms in convex programming, In: SIAM, vol. 13. (1994)
20. Bordin, M., Naks, T., Toom, A., Pantel, M.: Compilation of heterogeneous models: Motivations and challenges. In: ERTS. Société des Ingénieurs de l'Automobile, http://www.sia.fr (2012)
21. Izerrouken, N., Pantel, M., Thirioux, X.: Ssi Yan Kai, O.: Integrated formal approach for qualified critical embedded code generator. Formal Methods for Industrial Critical Systems. Lecture Notes in Computer Science, vol. 5825, pp. 199–201. Springer, Berlin (2009)
22. Toom, A., Izerrouken, N., Naks, T., Pantel, M., Ssi-Yan-Kai, O.: Towards reliable code generation with an open tool: Evolutions of the gene-auto toolset. In: ERTS. Société des Ingénieurs de l'Automobile, http://www.sia.fr (2010)

23. Toom, A., Naks, T., Pantel, M., Gandriau, M., Wati, I.: Gene-auto–an automatic code generator for a safe subset of simulink-stateflow and Scicos. In: ERTS. Société des Ingénieurs de l'Automobile, http://www.sia.fr (2008)
24. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. http://frama-c.cea.fr/acsl.html (2008)
25. Whalen, M.W., Murugesan, A., Rayadurgam, S., Heimdahl, M.P.: Structuring simulink models for verification and reuse. In: Proceedings of the 6th International Workshop on Modeling in Software Engineering, pp. 19–24. ACM (2014)
26. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c: A software analysis perspective. In: Proceedings of the 10th International Conference on Software Engineering and Formal Methods. SEFM'12, pp. 233–247. Springer, Berlin, Heidelberg (2012)
27. Scott, M.L.: Programming Language Pragmatics. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2000)
28. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**, 576–580 (1969)
29. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975)
30. Geneauto metamodel with verification annotations documentation. http://block-library.enseeiht.fr/html/Progress/geneautoAnnot.html
31. Jönsson, U.T.: A Lecture on the S-Procedure. Lecture Note at the Royal Institute of technology, Sweden (2001)
32. Owre, S., Shankar, N.: Theory interpretation in pvs. Techincal report, SRI International (2001)
33. Munoz, C.: Batch proving and proof scripting in pvs. NIA-NASA Langley. National Institute of Aerospace, Hampton, VA, Report NIA Report (2007)
34. Roux, P., Jobredeaux, R., Garoche, P.L., Feron, E.: A generic ellipsoid abstract domain for linear time invariant systems. In: HSCC, pp. 105–114 (2012)
35. Arthan, R., Martin, U., Oliva, P.: A hoare logic for linear systems. Formal Aspects Comput. **25**(3), 345–363 (2013)
36. Goubault, E., Putot, S.: A zonotopic framework for functional abstractions. CoRR abs/0910.1763 (2009)
37. dof helicopter: http://www.quanser.com/Products/3dof_helicopter
38. Price Induction: DGEN 380 turbofan engine. http://www.price-induction.com/en (2013)