Transferring Stability Proof Obligations from Model Level to Code Level

Michael Dierkes[1] and Daniel Kästner[2]

[1] Rockwell Collins France, 6 avenue Didier Daurat, 31701 Blagnac, France
mdierkes@rockwellcollins.com
[2] AbsInt GmbH, Science Park 1, 66123 Saarbrücken, Germany
kaestner@absint.com

1 Introduction

Bounded-input bounded-output (BIBO) stability is a fundamental requirement for filtering and signal processing systems, ensuring that the system output cannot grow indefinitely as long as the system input stays within a certain range. In general, it is necessary to identify and to prove some system invariant in order to prove BIBO stability. The objective of this work is to prove the BIBO stability of the floating point implementation in C of a signal processing unit which has been generated based on a specification in MATLAB Simulink©. This proof uses a system invariant which has been found by analyzing the Simulink model. A stability proof on code level, using a high confidence analysis tool like Astrée, is interesting since it might be taken into account for certification.

In model-based development, a model of a system is defined using a modeling language like Simulink before the possibly automated coding of the software is done. In order to ensure that the system behaves correctly, it can be used for simulation, and useful information like system invariants can be found using formal analysis or domain knowledge. As soon as the model is considered as sufficiently mature, the coding phase begins. Even if a correct automated coder is used, more analysis is in general necessary on the code level as the execution of the code may lead to runtime errors which are not detectable on the model level, or the precision of the computations may be lost due to accumulated rounding errors of floating point arithmetic.

The information which was obtained at the model level can be very useful for the code analysis, since it might be very difficult for code analysis tools to find them by themselves. Properties which hold on model level do not necessarily hold on code level when floating point numbers are used, but it is likely that a property which is only slightly different can be proven. For example, we might prove on model level that the absolute value of a variable x is always less than a constant C, i.e. $|x| < C$. Then, on the code level, it might hold that $|x| < C + \delta$ for a small δ which is due to the rounding error.

2 The RC Analysis Framework

For several years, Rockwell Collins has been developing and using a verification framework for MATLAB Simulink© and SCADE Suite™models which allows to translate these models into equivalent descriptions expressed in the input languages of different model checking and theorem proving tools. The usefulness of this industrial application of formal methods has been shown in different case studies [8], [9]. Thanks to recent advances in SMT solving, it is now possible to analyze aerospace domain models containing arithmetic computations on real variables. It is important to note that on model level, real arithmetic is assumed to have infinite precision, i.e. rounding errors or overflows are not considered.

The translation flow from MATLAB Simulink to different formal analysis languages is shown in figure 1. In order to develop its analysis capabilities, Rockwell Collins is closely working together with different research teams at the universities of Iowa and Minnesota, and the French *Office national d'études et de recherches aérospatiales* (ONERA).

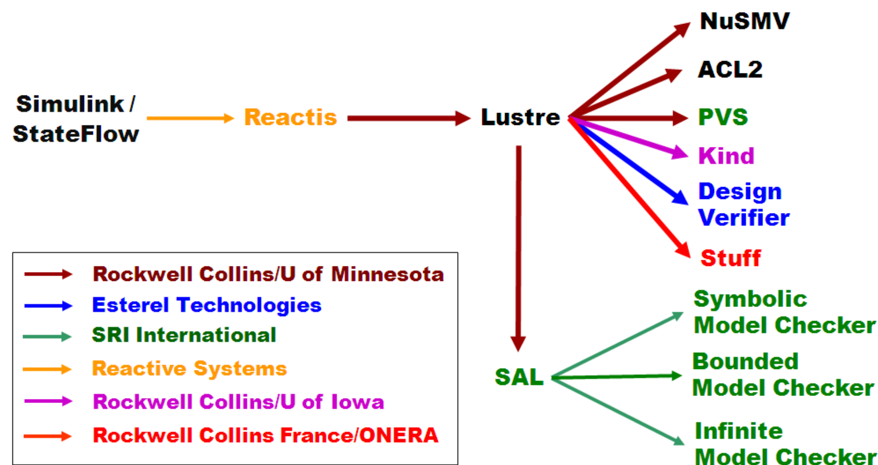

Figure 1. The translation flow of the Rockwell Collins analysis framework

The verification framework is used as a debugging tool: model checking may find erroneous behavior which requires the modification of the model. When all relevant properties are proven, the model can be considered as mature, and the coding phase can start in which the code might be generated automatically. However, the knowledge obtained by the model analysis (like for example invariants) is usually not exploited on the code level in any way, also because formal code analysis is not yet applied systematically. This might change thanks to the

availability of powerful code level analysis techniques like abstract interpretation, including support for qualification under relevant industry standards.

Currently, we are working on a technique to generate invariants automatically [4], which might significantly increase the degree of automatization of the model level analysis in our framework. This technique also benefits from recent advances in SMT solving and other areas like quantifier elimination. For a system similar to the case study presented in this article, the prototype implementation of our technique (called *Stuff*) was able to generate automatically the invariants which are necessary to prove the stability of the system.

3 The Astrée Tool

Astrée [2] is a parametric static analyzer based on abstract interpretation that aims at proving the absence of run-time errors of programs written in C, according to "ISO/IEC 9899:1999 (E)" (C99 standard) [3]. The class of errors reported includes out-of-bound array accesses, erroneous pointer manipulations and dereferencing, integer and floating-point division by zero, floating point overflows and invalid operations. Astrée also can address functional program properties by checking user-defined static assertions. In addition, Astrée warns about accesses to uninitialized variables and can detect code which is guaranteed to be unreachable. An important property of Astrée is that it takes floating-point rounding errors into account, conservatively assuming the worst-case of all possible rounding modes. This makes it very attractive to investigate the stability of floating-point algorithms.

For industrial use an important goal is to produce the fewest possible number of false alarms. An automatic proof of the absence of runtime errors is only possible if the analysis terminates without any alarm. Any alarm has to be manually checked by the developers – and this manual effort should be as low as possible. If there is a true error, it has to be fixed and the analysis should be restarted. A false alarm can possibly be eliminated by a suitable parametrization of Astrée: if the error cannot occur due to certain preconditions which are not known to Astrée, they can be made available to Astrée via analyzer directives. These directives make the side conditions explicit which have to be satisfied for a correct program execution. Furthermore, users can locally tune the precision of Astrée to analyze critical program parts with high precision, and improve speed by lowering the precision for uncritical program parts. This is further illustrated in Sec. 6.

4 The Case Study: Triplex Sensor Voter

The case study we used in this work is a triplex sensor voter, i.e. a redundancy management unit for three sensor input values. Our voter does not compute an average value, but uses the $MiddleValue(x, y, z)$ function, which returns the input value which is between the minimum and the maximum input values (for example, if $y < z < x$, it would return z). Other voter algorithms which use a

(possibly weighted) average value are more sensitive to one of the input values being out of the normal bounds. The Simulink model of the voter is shown in figure 2.

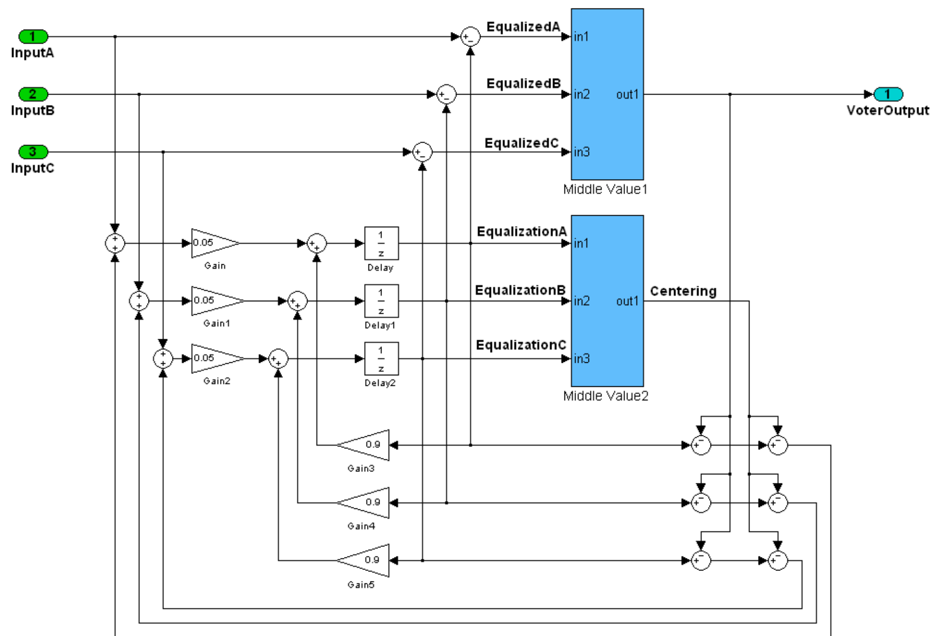

Figure 2. MATLAB Simulink model of the triplex sensor voter

A slightly more complicated version of this voter has been analyzed in [5]. It differs from the version presented in this article by the use of saturation operators which limit the change rate of the output.

The following recursive equations describe the behaviour of the voter:

$$
\begin{aligned}
EqualizationA_0 &= 0.0\\
EqualizationB_0 &= 0.0\\
EqualizationC_0 &= 0.0
\end{aligned}
$$

$$
\begin{aligned}
EqualizedA_t &= InputA_t - EqualizationA_t\\
EqualizedB_t &= InputB_t - EqualizationB_t\\
EqualizedC_t &= InputC_t - EqualizationC_t
\end{aligned}
$$

$$
\begin{aligned}
EqualizationA_{t+1} &= 0.9 * EqualizationA_t +\\
&\quad 0.05 * (InputA_t + ((EqualizationA_t - VoterOutput_t) - Centering_t))\\
EqualizationB_{t+1} &= 0.9 * EqualizationB_t +\\
&\quad 0.05 * (InputB_t + ((EqualizationB_t - VoterOutput_t) - Centering_t))\\
EqualizationC_{t+1} &= 0.9 * EqualizationC_t +\\
&\quad 0.05 * (InputC_t + ((EqualizationC_t - VoterOutput_t) - Centering_t))
\end{aligned}
$$

$$
Centering_t = middleValue(EqualizationA_t, EqualizationB_t, EqualizationC_t)
$$

$$
VoterOutput_t = middleValue(EqualizedA_t, EqualizedB_t, EqualizedC_t)
$$

5 Model Level Analysis

On model level, we want to prove the stability of the system, i.e. we want to prove that the voter output is bounded as long as the input values differ by at most the maximal authorized deviation $MaxDev$ from the true value of the measured physical quantity represented by the variable $TrueValue$. In our analysis, we fixed the maximal sensor deviation to 0.2, a value that domain experts gave us as typical value in practical applications.

The voter output is equal to one of the three equalized values, i.e.

$$
\begin{aligned}
VoterOutput &= EqualizedX\\
&= InputX - EqualizationX\\
&\leq (TrueValue + MaxDev) - EqualizationX
\end{aligned}
$$

with X in $\{A, B, C\}$. Therefore, if we can prove that the equalization values are bounded, the system must be stable. Furthermore, since the equalization values depend only on the differences between the input values and not on their absolute value, we can assume without loss of generality that the true value is 0.0. This means that if we can prove the stability for $TrueValue = 0.0$, then the system is also stable for all other possible values of $TrueValue$.

As mentioned before, we assume $MaxDev = 0.2$, so we constrain the input values by

$$
|InputA| \leq 0.2 \text{ and } |InputB| \leq 0.2 \text{ and } |InputC| \leq 0.2
$$

Using a model checker like for example Kind [7], we can prove by induction that the following expression is an inductive invariant:

$$
|EqualizationA| < 0.4 \text{ and } |EqualizationB| < 0.4 \text{ and } |EqualizationC| < 0.4
$$

The value of 0.4 can be found by trial and error, or automatically using the invariant generation approach described in [5]. In general, it turned out that the absolute value of the equalization variables is bounded by $2 * MaxDev$.

Note that for this version of the voter, no further invariants are necessary for the stability proof. This is different in the case of the voter analyzed in [5], where additional invariants are necessary which give an upper bound for the pairwise sum resp. difference of the equalization values. However, it is likely that these invariants can be generated automatically by future versions of our analysis tool *Stuff*.

6 Code Level Analysis

The code we analyzed has been generated automatically from the Simulink model. The three input sensor values are assumed to be fully volatile with ranges between -0.2 and $+0.2$. This can be directly represented by Astrée directives:

```
__ASTREE_volatile_input((SensorA, [-0.2, 0.2]));
__ASTREE_volatile_input((SensorB, [-0.2, 0.2]));
__ASTREE_volatile_input((SensorC, [-0.2, 0.2]));
```

Our proof obligation is formulated by Astrée assertions:

```
__ASTREE_assert((state.Eq1_memory <= 0.4));
__ASTREE_assert((state.Eq1_memory >= -0.4));
__ASTREE_assert((state.Eq2_memory <= 0.4));
__ASTREE_assert((state.Eq2_memory >= -0.4));
__ASTREE_assert((state.Eq3_memory <= 0.4));
__ASTREE_assert((state.Eq3_memory >= -0.4));
```

In contrast to dynamic assertions that are checked during program runtime, Astrée assertions are checked statically. If Astrée does not report a violation of such an assertion, its correctness has been formally proven. While the focus of Astrée is on non-functional properties – absence of runtime errors – the assertion mechanism makes it possible to also prove functional program properties by an Astrée analysis.

The main function contains an infinite loop which reads the sensor inputs and executes the voting algorithm which are encapsulated in an own function `voter_nosat2_compute`:

```
int main(){
  voter_nosat2_init(&state);
  while(1)
  {
    io.In1 = SensorA;
    io.In2 = SensorB;
    io.In3 = SensorC;
```

```
    __ASTREE_assert((state.Eq1_memory <=  0.4));
    __ASTREE_assert((state.Eq1_memory >= -0.4));
    __ASTREE_assert((state.Eq2_memory <=  0.4));
    __ASTREE_assert((state.Eq2_memory >= -0.4));
    __ASTREE_assert((state.Eq3_memory <=  0.4));
    __ASTREE_assert((state.Eq3_memory >= -0.4));
    voter_nosat2_compute(&io, &state);
  }
}
```

The implementation of the function `voter_nosat2_compute` is shown below in abbreviated form (variable declarations have been omitted):

```
void voter_nosat2_compute(t_voter_nosat2_io *_io_,
                          t_voter_nosat2_state *_state_) {
    /* Variable declarations */
    In1 = _io_->In1; In2 = _io_->In2; In3 = _io_->In3;
    Constant = 0.05;
    voter_nosat2_Eq1 = _state_->Eq1_memory;
    Sum = In1 - voter_nosat2_Eq1;
    voter_nosat2_Eq2 = _state_->Eq2_memory;

    Sum1 = In2 - voter_nosat2_Eq2;
    Relational_Operator_1 = voter_nosat2_Eq1 > voter_nosat2_Eq2;
    Relational_Operator_2 = Sum > Sum1;

    voter_nosat2_Eq3 = _state_->Eq3_memory;
    Sum2 = In3 - voter_nosat2_Eq3;

    Relational_Operator2_2 = Sum2 > Sum;
    Relational_Operator2_1 = voter_nosat2_Eq3 > voter_nosat2_Eq1;
    Relational_Operator1_2 = Sum1 > Sum2;
    Relational_Operator1_1 = voter_nosat2_Eq2 > voter_nosat2_Eq3;

    Relational_Operator3_1 =
        Relational_Operator_1 == Relational_Operator1_1;
    Relational_Operator3_2 =
        Relational_Operator_2 == Relational_Operator1_2;

    Relational_Operator4_1 =
        Relational_Operator1_1 == Relational_Operator2_1;
    Relational_Operator4_2 =
        Relational_Operator1_2 == Relational_Operator2_2;

    if (Relational_Operator4_2 >= 1) {
        Switch_2 = Sum2;
    } else {
```

```
        Switch_2 = Sum;
    }

    if (Relational_Operator3_2 >= 1) {
          Switch1_2 = Sum1;
     } else {
        Switch1_2 = Switch_2;
    }

    _io_->VoterOutput = Switch1_2;

    if (Relational_Operator4_1 >= 1) {
        Switch_1 = voter_nosat2_Eq3;
    } else {
        Switch_1 = voter_nosat2_Eq1;
    }

    if (Relational_Operator3_1 >= 1) {
        Switch1_1 = voter_nosat2_Eq2;
    } else {
        Switch1_1 = Switch_1;
    }

    Product5 = 0.9 * voter_nosat2_Eq2;
    Sum4 = voter_nosat2_Eq2 - Switch1_2;
    Sum7 = Sum4 - Switch1_1;
    Sum13 = In2 + Sum7;
    Product2 = Constant * Sum13;
    voter_nosat2_Sum10 = Product2 + Product5;
    Product6 = 0.9 * voter_nosat2_Eq3;
    Sum5 = voter_nosat2_Eq3 - Switch1_2;
    Sum8 = Sum5 - Switch1_1;
    Sum14 = In3 + Sum8;
    Product3 = Constant * Sum14;
    voter_nosat2_Sum11 = Product3 + Product6;
    _state_->Eq3_memory = voter_nosat2_Sum11;
    _state_->Eq2_memory = voter_nosat2_Sum10;
    Sum3 = voter_nosat2_Eq1 - Switch1_2;
    Sum6 = Sum3 - Switch1_1;
    Sum12 = In1 + Sum6;
    Product1 = Constant * Sum12;
    Product4 = 0.9 * voter_nosat2_Eq1;
    voter_nosat2_Sum9 = Product1 + Product4;
    _state_->Eq1_memory = voter_nosat2_Sum9;
}
```

To enable a high analysis precision it is important to precisely analyze the condional statements where the variables Switch_1, Switch_2, and Switch1_2 are defined. While Astrée can automatically perform partitioning of if-statements such that the effects of the then- and else-parts are disambiguated during the further analysis of the program, the conditions already encoded in the Relational_Operator variables are not automatically disambiguated.

An important design feature of Astrée is that it allows users to tune the precision of the analysis to the software under analysis at a fine-grained level. One such mechanism is the __ASTREE_partition_control directive which can instruct Astrée to partition also on the values of a boolean variable. This directive is inserted for all the definitions of the Relational_Operator variables as shown in the following example:

```
__ASTREE_partition_control Relational_Operator_2 = Sum > Sum1;
```

While this transformation has been done manually in our experiment, it could easily be done by the code generator or a pre-peprocessor in an automated way. Furthermore it is important to note that the Astrée directives do not have to be placed in the generated code, but can also be specified externally without modifying the generated code. Externally specified directives are relocated by using a dedicated annotation language AAL, which allows to specify any specific program point but does not rely on line number information [1].

When running the analysis on the transformed code, the inserted assertions fail. The value analysis of Astrée shows that due to rounding errors the computed variable ranges can be slightly above 0.4, resp., below -0.4. Based on the reported rounding error we replaced the bounds by 0.400001, resp., -0.400001. After this modification, the Astrée analysis terminates with *zero alarms*, yielding a proof of absence of runtime error, and a proof of the bounds on the equalization values.

7 Conclusion and Future Work

We were able to prove an invariant that allows to conclude for bounded-input bounded-output stability of a redundancy management unit. This invariant was found on model level using model checking, and then transfered to the automatically generated code. From a methodological perspective this process is a combination of results of model-checking and abstract interpretation. Invariants which have been proven at the modeling level by model-checking are transferred to the code level and mapped to static assertions on value ranges which are amenable to abstract-interpretation-based static analysis. The static analysis done by Astrée allows to find new invariants which hold at the C code level and take potential rounding errors into account.

Model checking and abstract interpretation are techniques with different scopes. In general, we cannot hope that properties which have been proven by model checking may be confirmed by abstract interpretation. However, we are looking for the most useful way to combine these techniques. In the course of the

ARTEMIS project MBAT we are planning to further generalize and automate this approach.

We intend to apply this approach on control systems. However, the use of model checking tools may be limited since such systems are typically nonlinear. In this case, invariants would be found based on control theory as described in [6].

Acknowledgement

We would like to thank Antoine Miné, Jérôme Feret and Xavier Rival from the École Normale Supérieure in Paris for supporting us in the Astrée analysis of the voter software.

References

1. AbsInt GmbH. The Static Analyzer Astrée. User Documentation for AAL Annotations, Dec. 2011.
2. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207, San Diego, California, USA, June 7–14 2003. ACM Press.
3. JTC1/SC22. Programming languages – C, 16 Dec. 1999.
4. A. Champion, R. Delmas, and M. Dierkes. Backward property directed reachability analysis based on quantifier elimination. 2011. Submitted for publication.
5. M. Dierkes. Formal analysis of a triplex sensor voter in an industrial context. In G. Salaün and B. Schätz, editors, *Proceedings of the 16th International Workshop on Formal Methods for Industrial Critical Systems, FMICS 2011*, volume 6959 of *LNCS*. Springer, 2011.
6. E. Feron. From Control Systems to Control Software. *IEEE Control Systems Magazine*, 30(6):50–71, Dec. 2010.
7. G. Hagen and C. Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In A. Cimatti and R. Jones, editors, *Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design (Portland, Oregon)*, pages 109–117. IEEE, 2008.
8. S. P. Miller, M. W. Whalen, and D. D. Cofer. Software model checking takes off. *Commun. ACM*, 53(2):58–64, 2010.
9. M. W. Whalen, D. D. Cofer, S. P. Miller, B. H. Krogh, and W. Storm. Integration of formal analysis into a model-based software development process. In *FMICS*, pages 68–84, 2007.