Robustness of Simulink/Stateflow Model Against Implementation Imperfections

Meng Li, Member IEEE * Ratnesh Kumar, Fellow IEEE **

* Dept. of Electrical and Computer Engineering, Iowa State University, Ames, IA 50010, USA (e-mail: mengl@iastate.edu)
** Dept. of Electrical and Computer Engineering, Iowa State University, Ames, IA 50010, USA (e-mail: rkumar@iastate.edu)

Abstract: Model-based software development frameworks such as Simulink/Stateflow support auto code generation. Due to the limitations of the implementation platform on which the generated code is deployed on, imprecision is introduced to the implementation and may lead to unpredictable behaviors in the implementation. In this paper, an implementation model is defined to model the imprecisions introduced by the platform. We present a notion of Path-Robustness (P-Robustness) and Path/Output-Robustness (P/O-Robustness) between the software model and its implementation model to determine if the implementation preserves the control and data flow of the software. An approach is proposed to check the P-Robustness and P/O-Robustness properties of the software model by constructing an error propagation model from the implementation model. P/O-Robustness is proved stronger than approximate bisimulation introduced by Girard et al.

Keywords: Robustness, Implementation, Imprecision, Simulink, Extended Finite Automata.

1. INTRODUCTION

The hardware platforms that execute software have limited computation power so signal values may get perturbed due to finite precision arithmetic, and the implemented system may perform different behaviors than that feasible in the software. A good software development framework should consider the platform imprecisions and the signal perturbations to avoid the unpredictable behaviors in the implementation, i.e. the implementation should preserve the control and data flow of the Simulink/Stateflow model. In this way, possible failures can be avoided at the early stages of the design process.

Robust verification/testing is a technique to verify the correctness of the implementation by capturing the platform level inaccuracies. It has been studied in the setting of timed automata where the goal is to examine whether the implementation of a timed-automaton will preserve its safety/temporal properties: Wulf et al. (2004) proposed the Almost ASAP (AASAP) semantics for timed automata to overcome the implementability problems by capturing the clock drifts and controller reaction delay; Fainekos and Pappas (2006) defined the robust satisfaction of Metric Temporal Logic (MTL) formulas for the continuous or discrete-time signals with perturbation. Potop-Butucaru et al. (2006) studied the effect of asynchronous communication presence in the implementation of a synchronous design. Xu and Kumar (2008) introduced the notion of bounded-delay implementability and an algorithm of checking the satisfiability of the property. Xu et al. (2010) studied the effect of finite precision measurement of time on control and diagnosis, respectively, of real-time systems.

The deployment of the software onto a computing hardware platform leads to some implementation effects, such as finite precision computation, computation/communication delay, and signal perturbation. Model-based Verification & Validation methods should capture such platform level inaccuracies, so that the design error caused by the implementation effects can be detected at an earlier stage.

The robust verification seeks the following property: Whenever a model $M$ satisfies property $\Psi$, then it also holds that its implementation $I(M)$ also satisfies the property $\Psi$. To formulate this problem one needs to define the semantics of the model $M$ (which is the definition of all of its runs) and also the semantics of the implementation $I(M)$ (which will define the runs of $I(M)$). Alternatively, one can directly define the model $I(M)$ for the implementation.

In our previous works described by Zhou and Kumar (2009, 2010); Li and Kumar (2011, 2013a), we introduced a recursive method to translate the software written in the model-based framework of Simulink/Stateflow diagram to an Input/Output Extended Finite Automaton (I/O-EFA), which makes it amenable to formal analysis. In this paper, in order to capture the platform level inaccuracies, an I/O-EFA model of a software is extended with finite precision semantics, and signal perturbation to form an implementation model. The concept of Path-Robustness (P-Robustness) and Path/Output-Robustness (P/O-Robustness) is introduced to describe the property of the software models to preserve control and data flow under certain inaccuracies introduced by the implementation. P/O-Robustness (that preserves both control and data flow) is proved to be stronger than approximate bisimulation described by Girard and Pappas (2011).

* The research was supported in part by the National Science Foundation under the grants NSF-ECCS-0801763, NSF-ECCS-0926029, and NSF-CCF-1331390.
While approximate bisimulation is all that is needed to preserve input-output computations to within a tolerance, a stronger condition of path-robustness is required for preserving control flow to guarantee path dependent properties such as path-coverage offered by a set of tests.

We present an approach to check the P-Robustness and P/O-Robustness by constructing an error propagation model from the implementation model. The error propagation model propagates the errors along the model evolutions and identifies the occasions when software violates the P-Robustness and P/O-Robustness properties. P-Robustness and P/O-Robustness checking is reduced to the problem of checking the reachability of a fault-location in the error propagation model. We also prove that P/O-Robustness is a sufficient condition of approximate bisimulation. The contributions of the paper are:

- Given an automaton model of a software, we present a method to augment the model so as to capture the semantics of the implementation, which may differ from the semantics of the original model owing to inaccuracies (e.g., finite precision arithmetic) introduced by the implementation.
- We present a notion of P-Robustness and P/O-Robustness to describe whether the implementation preserves the control and data flow of the software model. P/O-Robustness is proved to be stronger than approximate bisimulation; the latter only guarantees output-robustness (that preserves data flow) and not path-robustness (that preserves control flow).
- An approach for checking P-Robustness and P/O-Robustness is presented in form of a reachability checking problem.

2. INTRODUCTION TO I/O-EFA

An I/O-EFA is a symbolic description of a reactive untimed infinite state system in form of an automaton, extended with discrete variables of inputs, outputs and data.

**Definition 1.** An I/O-EFA is a tuple

\[ P = (L, D, U, Y, \Sigma, \Delta, L_0, D_0, L_m, E), \]

where

- \( L \) is the set of locations (symbolic-states),
- \( D = D_1 \times \ldots \times D_n \) is the set of data (numeric-states),
- \( U = U_1 \times \ldots \times U_m \) is the set of numeric inputs,
- \( Y = Y_1 \times \ldots \times Y_p \) is the set of numeric outputs,
- \( \Sigma \) is the set of symbolic-inputs,
- \( \Delta \) is the set of symbolic-outputs,
- \( L_0 \subseteq L \) is the set of initial locations,
- \( D_0 \subseteq D \) is the set of initial-data values,
- \( L_m \subseteq L \) is the set of final locations,
- \( E \) is the set of edges, and each \( e \in E \) is a 7-tuple,
  
  \[ e = (a_e, t_e, \sigma_e, \delta_e, G_e, f_e, h_e), \]

  where

  - \( a_e \in L \) is the origin location,
  - \( t_e \in L \) is the terminal location,
  - \( \sigma_e \in \Sigma \cup \{ \varepsilon \} \) is the symbolic-input,
  - \( \delta_e \in \Delta \cup \{ \varepsilon \} \) is the symbolic-output,
  - \( G_e \subseteq D \times U \times Y \) is the enabling guard (a predicate),
  - \( f_e : D \times U \rightarrow D \) is the data-update function, and
  - \( h_e : D \times U \rightarrow Y \) is the output-assignment function.

I/O-EFA \( P \) starts from an initial location \( l_0 \in L_0 \) with initial data \( d_0 \in D_0 \). When at a state \((l, d)\), a transition \( e \in E \) with \( a_e = l \) is enabled, if the input \( \sigma_e \) arrives, and the data \( d \) and input \( u \) are such that the guard \( G_e(d, u) \) holds. \( P \) transitions from location \( a_e \) to location \( t_e \) through the execution of the enabled transition \( e \) and at the same time the data value is updated to \( f_e(d, u) \), whereas the output variable is assigned the value \( h_e(d, u) \) and a discrete output \( \delta_e \) is emitted. In what follows below, the data update and output assignments are performed together in a single action.

3. MODELING THE IMPLEMENTATION

**IMPRECISION**

Imprecision may be introduced when the generated code is executed on a platform with computational limitations, such as:

- Computing platforms can only represent a limited range of values. The computed and represented results are rounded to the nearest representable value, leading to the floating point representation error and floating point arithmetic error.
- Sensors, actuators, and communication components of the system may introduce perturbations leading to signal perturbation error.

Although errors are small when they are introduced, they can be accumulated during the course of the execution, through the feedback loops. The accumulated error can affect the data flow as well as the control flow of the system.

**Example 1.** Consider the Simulink model (model-based development tool provided by Simulink/Stateflow (1984)) of a bounded counter shown in Figure 1, consisting of an enabled subsystem block, a saturation block, and a rounding block. The output \( y_5 \) increases by 1 at each sample-period when the control input \( u \) is positive, and \( y_5 \) resets to its initial value when the control input \( u \) is not positive. The saturation block limits the value of \( y_2 \) within the range between \(-0.5 \) and \( 20 \). The rounding block rounds the value of \( y_2 \) to nearest integer. The incremental and summation operation have a maximum error of \( \pm 0.01 \), and input signal has a maximum perturbation of \( \pm 0.01 \). Then each execution of the feedback loop incurs a maximum error of \( \pm 0.02 \), i.e., 25 executions of the feedback loop would lead to a maximum error of \( \pm 0.5 \), which may affect the rounding result for \( y_6 \).

![Fig. 1. Simulink diagram of a counter.](image-url)
The imprecision level introduced by the implementation is determined by the implementation configuration, such as the choice of computing systems, sensors/actuators, communication components, etc. In order to quantify the errors introduced from the implementation configuration, we map the implementation configuration to a set of errors, which are the maximum deviations from the true values introduced by the configuration. We let \( \epsilon_v \) to denote the perturbed variable with the maximum deviation \( \epsilon \) from the true value \( v \). The arithmetic of the perturbed variables is shown in Table 1. For example, using Table 1 we have:

\[
(\varepsilon_1 \times \varepsilon_2) + (\varepsilon_3 + \varepsilon_4) = (\varepsilon_1 + \varepsilon_2) \times (\varepsilon_3 + \varepsilon_4) - (\varepsilon_1 \times \varepsilon_2) - (\varepsilon_3 - \varepsilon_4) + (\varepsilon_2 \times \varepsilon_3)
\]

We model the imprecision of the implementation by introducing the perturbed variables to each of the variables and perturbed arithmetic to each of the data-update and output-assignment functions in the I/O-EFA model, based on the set of errors that the implementation configuration introduces. The construction of the implementation model with regard to an implementation configuration is computed in the following algorithm.

**Algorithm 1.** Given an I/O-EFA model

\[ P = (L, D, U, Y, \Sigma, \Delta, L_0, D_0, L_m, E) \]

\[ P^f \] under implementation configuration \( I \) is a tuple \( \hat{P} = (\hat{L}, \hat{D}^f, \hat{U}^f, \hat{Y}^f, \Sigma, \Delta, \hat{L}_0, \hat{D}_0^f, \hat{L}_m, E^f) \), where

\[
\begin{align*}
\hat{L}^f &= L \times \{d \in D \mid [d] = d \} \text{ is the perturbed } D \text{ under } I, \\
\hat{U}^f &= U \times \{u \in U \mid [u] = u \} \text{ is the perturbed } U \text{ under } I, \\
\hat{Y}^f &= Y \times \{y \in Y \mid [y] = y \} \text{ is the perturbed } Y \text{ under } I, \\
\hat{D}_0^f &= \{d_o \in D \mid d_o = d \} = \text{ the perturbed } D_0 \text{ under } I, \\
E^f &= \text{ the implementation of } E \text{ under } I, \text{ and for each } e = (\sigma, l, \delta, G_e, f_e, h_e) \in E, \text{ its implementation } e^f = (\sigma, l, \delta, G_{e^f}, f_{e^f}, h_{e^f}) \in E^f, \text{ where}
\end{align*}
\]

\[
\begin{align*}
G_{e^f} &= G_e([d]_e, [u]_e) \text{ is the weakened guard of } G_e, \\
f_{e^f} &= [f_e]([d]_e, [u]_e), \text{ is the perturbed } f_e \text{ with perturbed variables under } I, \\
h_{e^f} &= [h_e]([d]_e, [u]_e), \text{ is the perturbed } h_e \text{ with perturbed variables and arithmetic under } I.
\end{align*}
\]

Note that our approach renders the implementation model \( P^f \) to have the same discrete structure as the software model \( P \).

**Example 2.** Consider the Simulink model in Example 1 as shown in Figure 1. The translated I/O-EFA model \( P \) (shown in Figure 2) is obtained by applying the approach described by Zhou and Kumar (2010) on the Simulink model in Figure 1. According to Algorithm 1, the implementation model \( P^f \) (shown in Figure 3) can be obtained by considering the implementation configuration, i.e., inaccuracies mentioned in Figure 1.

4. **ROBUSTNESS OF THE SOFTWARE**

Engineers expect the implementations to behave the same as the software, i.e., the implementation should preserve control flow (the computation path executed on a certain input sequence) and the data flow (output values computed along the computation path). We define Path-Robustness (P-Robustness) and Path/Output (P/O-Robustness) to determine whether the implementation configuration changes the control or data flow behaviors of the software model.

**Definition 2.** A software model \( P \) is Path-Robust (or simply P-Robust) under an implementation configuration \( I \), if each input sequence applied on its I/O-EFA model \( P \) and implementation model \( P^f \) produce the same edge-sequences (preserves control flow). \( P \) is Path/Output-Robust (or simply P/O-Robust) under \( I \), if each input sequence applied on its I/O-EFA model \( P \) and implementation model \( P^f \) produce the same edge-sequences and the two output sequences are within tolerance bound (preserves control and data flow).

P-Robustness requires implementation model \( P^f \) to activate the same edge-sequences as I/O-EFA model \( P \) activates under any sequence of inputs. Since at each location in \( P \), the disjoint of the guards of its outgoing edges is the entire data-input space, the weakened guards in \( P^f \) may introduce nondeterministic edge choices at each location. If a nondeterministic edge choice is enabled in \( P^f \), then under some input sequence \( P^f \) may activate a different edge sequence from the one activated by \( P \), thereby violating P-Robustness. Therefore, verification of P-Robustness is reduced to checking if nondeterministic edge choices in \( P^f \) are reachable. P/O-Robustness additionally requires the output deviations (errors of the perturbed variables in \( P^f \)) to be within user-specified tolerance limits. Therefore, P/O-Robustness can be reduced to checking P-Robustness plus whether the perturbed output variables have error less than the tolerance. We first introduce the notions of deterministic and nondeterministic regions for each edge by partitioning the corresponding edge guards as follows.

**Definition 3.** Given a location \( l \in L \) and an edge \( e \in \text{ edge}(l) := \{e \in E \mid \sigma_o = l\} \) in implementation model \( P \), the deterministic region of the guard of edge \( e \) is \( G_e^d := (\Lambda_{e' \in \text{ edge}(l), e' \neq e} G_{e'}^d) \cap G_e^d \), and the nondeterministic region is \( G_e^n := \bigvee_{e' \in \text{ edge}(l), e' \neq e} G_{e'}^d \cap G_e^d \), where

- \( G_e^d(d, u, y, \epsilon(d), \epsilon(u), \epsilon(y)) := \exists ([d]_e, [u]_e, [y]_e) : G_e^d([d]_e, [u]_e, [y]_e) \wedge \Lambda_{e' \in \text{ edge}(l), e' \neq e} \epsilon(d') \leq \epsilon(d) \) is the relaxation of the edge guard \( G_e \) by projecting \([d]_e, [u]_e, [y]_e\) to \(d, u, y, \epsilon(d), \epsilon(u), \epsilon(y)\).

Upon partitioning, the deterministic region does not have overlap with the guards of other edges originating at the same location while the nondeterministic region has overlap with the guards of other edges originating at the same location.
In order to verify the P-Robustness and P/O-Robustness, an error propagation model is computed below to check if nondeterministic regions are reachable and output is within tolerance. The error propagation model tracks two dynamics for each variable: One for the variable itself as in the original model, and the other for its error. The verification step also introduces a new fault location $f$ which is reached without (resp., with) constraint on tolerance bound for output error if and only if P-Robustness (resp., P/O-Robustness) is violated.

Algorithm 2. Given an implementation model $P^I = (L, D^I, U^I, Y^I, \Sigma, \Delta, L_0, D_0^I, L_m, E^I)$ of $P$ under implementation configuration $I$ and an output tolerance $\sigma$, the error propagation model is a tuple $P^{error} = (L \cup \{f\}, D^{error}, U^{error}, Y^{error}, \Sigma, \Delta, L_0, D_0^{error}, L_m, E^{error})$, where

- $D^{error} = \bigcup_{d^I \in D^I} \{l_d^I, e_d^I\}$ is the set of augmented data variables and the error variables of the perturbed data,
- $U^{error} = \bigcup_{u^I \in U^I} \{l_u^I, e_u^I\}$ is the set of augmented data variables and the error variables of the perturbed inputs,
- $Y^{error} = \bigcup_{y^I \in Y^I} \{l_y^I, e_y^I\}$ is the set of augmented data variables and the error variables of the perturbed outputs,
- $D_0^{error} = \bigcup_{d_0^I \in D_0^I} \{l_{d_0^I}, e_{d_0^I}\} \bigcup_{e_u^I \in E^I} \{e_{u^I}\}$ is the set of augmented data variables and the error variables of the perturbed initial data and error of the initial inputs, and
- $E^{error} = E^d \cup E^f$ is the set of edges, where $E^d$ is the set of error propagation edges with deterministic guards, and for each
\[ e^f = (l, t_e, \sigma_e, \delta_e, G^d_e, f_e^d, h_e^d) \in E^f \]
its error
property
e edge \( e^{d} = (o^{d}, t^{d}, \sigma^{d}, \delta^{d}, G^{d} \wedge \left[ e^{d}(Y^{d}) \leq \sigma\right], f^{d}, h^{d}_{d} \in E^{d} \) where
- \( G^{d}_{d} \) is the deterministic region of \( G^{d} \).
- \( f^{d} = \{ (l, f^{d}), \sigma(h^{d}) \} \) is the true value update function and its error propagation function, and
- \( h^{d}_{d} = \{ (l, h^{d}), \sigma(h^{d}) \} \) is the true value output-assignment function and its error propagation function;
\[ E^{f} = \bigcup_{e \in E^{f}} \forall e \in \text{edge}(l) \land G^{d}_{e} = \text{False} \]
\[ \{ (l, f, -, -) \land \forall e \in \text{edge}(l) \land G^{d}_{e} = \text{True} \}
\]
The above error propagation model partitions each edge guard to deterministic and nondeterministic regions. When nondeterministic regions or intolerable errors are reached, location \( f \) is reached and either P-Robustness (tolerance \( \sigma = \infty \)) or P/O-Robustness (tolerance \( \sigma \) bounded by a user-specified value) is violated.

**Example 3.** Consider the model in Example 1 as shown in Figure 1 and its implementation model in Figure 3 with counter output tolerance 0.5. According to Algorithm 2, the error propagation model can be obtained as shown in Figure 4.

The following theorem establishes that P-Robustness and P/O-Robustness checking is converted to fault-location reachability checking in the error propagation model. The proof is clear through the construction of the error propagation model.

**Theorem 1.** A software model is P-Robust under an implementation configuration \( I \) if and only if location \( f \) is not reachable in its error propagation model \( P^{err} \) under \( \sigma = \infty \) (i.e., disregarding the constraint on tolerance).

Similarly, \( P \) is P/O-Robust under \( I \) if and only if location \( f \) is not reachable in \( P^{err} \).

**Example 4.** Consider the error propagation model (shown in Figure 4) obtained in Example 3 for the model in Figure 1 and its implementation model in Figure 3 with counter output tolerance 0.5. After checking reachability of fault-location \( f \) in the error propagation model with \( \sigma := 0.5 \), \( f \) is reachable with location trace \( (l_{0}^{f}, f) \) when \( u(0) = 0 \), since at \( l_{0}^{f} \) guard \((-0.01 < u(0) \leq 0.01) \lor (u(0) > 0.01 \land 0 > 0) \) is true and the error propagation model transits to \( f \). (Input \( u(0) = 0 \) with perturbation \( \pm 0.01 \) can either disable or enable the subsystem.) Therefore, P/O-Robustness is violated.

We next compare P/O-Robustness with the notion of approximate bisimulation described by Girard and Pappas (2011) and show that the former is a stronger notion. A brief description of approximate bisimulation for I/O-EFA models is presented below.

For an I/O-EFA \( P \), the notation \( (l, d)^{\sigma, \delta, u, y}(l', d') \) implies the existence of \( e \in E \) such that \( o = l, l_{e} = l' \), \( \sigma = \sigma, \delta_{e} = \delta, G_{e}(d, u) \) holds, and \( d = f_{e}(d, u) \) and \( y = h_{e}(d, u) \).

**Definition 4.** Given an I/O-EFA \( P \), a approximate simulation relation over its states of precision \( \varepsilon \) is a binary relation \( \Phi_{\varepsilon} \subseteq (L \times D) \times (L \times D) \) such that \((l_{1}, d_{1}), (l_{2}, d_{2}) \in \Phi_{\varepsilon} \) implies \( \forall e \in L, u_{e}, \exists e_{2} : e_{2} = e_{2} \iff \sigma_{e_{2}} = \sigma_{e} \land d_{e_{2}} = d_{e} \land \left[ e_{2}(Y_{e_{2}}) \leq \sigma_{e} \right] \land \left[ e_{2}(D_{e_{2}}) = d_{e} \right] \right) \Rightarrow \exists (l_{2}, d_{2})^{\sigma_{e_{2}}, u_{e_{2}}, y_{e_{2}}}(l_{2}, d_{2}) \such \that ((l_{1}, d_{1}, (l_{2}, d_{2})) \in \Phi_{\varepsilon} \land \left[ y_{e_{2}} - y'_{e_{2}} \right] \leq \varepsilon \).

A symmetric approximate simulation relation is called approximate bisimulation relation. Two systems \( P_{1} \) and \( P_{2} \) are said to be approximate bisimilar if there exists a approximate bisimulation relation \( \Phi_{\varepsilon} \subseteq (L_{1} \times D_{1}) \times (L_{2} \times D_{2}) \) such that for each \((l_{10}, d_{10}) \in L_{1} \times D_{1} \) there exists \((l_{20}, d_{20}) \in L_{2} \times D_{2} \) such that \((l_{10}, d_{10}, (l_{20}, d_{20})) \in \Phi_{\varepsilon} \).

**Theorem 2.** An I/O-EFA model \( P \) is approximate bisimilar to its implementation model \( P^{I} \) with tolerance \( \sigma \) whenever \( P \) is P/O-Robust under the implementation configuration \( I \) with tolerance \( \sigma \). The converse need not hold in general.

**Proof:** Since \( P^{I} \) preserves the control and data flow within tolerance bound \( \sigma \) of I/O-EFA model \( P \), an approximate bisimulation relation between states of \( P \) and \( P^{I} \) can be found by pairing states of \( P \) and \( P^{I} \) that are visited on runs of identical inputs. (Note these runs preserve the location sequence visited, and the outputs at each step differ no more than the tolerance bound.) To see that the converse is not necessarily true, consider the counterexample in Figure 5. At location \( l_{1}^{f} \) when \( u = 1 \) implementation model (right) can activate both \( e_{0} \) and \( e_{1} \) but I/O-EFA (left) can activate only \( e_{1} \), so P/O-Robustness is violated; however, since both \( e_{0} \) and \( e_{1} \) lead to location \( l_{1}^{f} \) and the values of output \( y \) of \( e_{0} \) and \( e_{1} \) are within tolerance \( \sigma = 0.5 \), the two models are approximate bisimilar with tolerance \( \sigma = 0.5 \).

**Fig. 5.** An I/O-EFA (left) and its implementation model (right) violating P/O-Robustness but satisfying approximate bisimulation.

The overall algorithm for the P-Robustness and P/O-Robustness checking is as the algorithm below.

**Algorithm 3.** 1. Translate software to I/O-EFA model \( P \) (e.g., software designed with Simulink/Stateflow can be translated following the approaches described by Zhou and Kumar (2010); Li and Kumar (2011, 2013a)).
2. According to Algorithm 1, construct implementation model \( P^{I} \) under implementation configuration \( I \).
3. According to Definition 3, identify the deterministic and nondeterministic regions of each edge in \( P^{I} \) for construction of error propagation model.
4. According to Algorithm 2, translate \( P^{I} \) to the error propagation model \( P^{err} \) with infinite (resp., specified) output tolerance for P-Robustness (resp., P/O-Robustness) checking.
5. Check reachability of fault-location $f$ in the error propagation model. If $f$ is reachable, P-Robustness (resp., P/O-Robustness) is violated; else, P-Robustness (resp., P/O-Robustness) holds. (Note the reachability checking in I/O-EFA models can be performed using model checking, constraint solving, or reachability resolution techniques proposed in our prior works described by Li and Kumar (2012, 2013b).)

5. CONCLUSION

The paper presented the notions of P-Robustness and P/O-Robustness for software models against implementation configuration so as to preserve control/data flows (precisely/within tolerance) in face of inaccuracies introduced by the implementation platform such as finite precision arithmetic. In order to check the P-Robustness and P/O-Robustness properties, an implementation model was introduced to capture the imprecisions caused by the implementation configuration, and this was then used to construct the error propagation model capturing the error propagation. Checking P-Robustness and P/O-Robustness was reduced to the reachability analysis of a fault-location in the error propagation model. P/O-Robustness was shown to be stronger than the notion of approximate bisimulation that ensures robustness of output, but is not strong enough to assure the robustness of control flow (so path-coverage property of test cases may not be preserved under approximate bisimulation).

REFERENCES


