Real-Time Reachability Monitor for Copilot

Ryan Spring

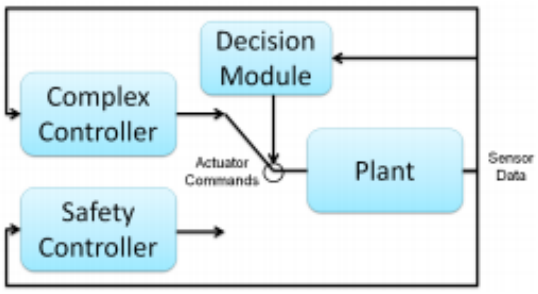

Fig. 1. Simplex Architecture

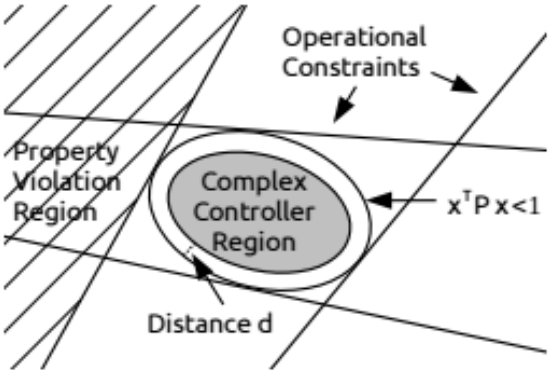

Fig. 2. The LMI Simplex design approach uses switching logic based on an ellipsoid within the system constraints in order to produce a verified system

I. Simplex Architecture

The Simplex Architecture ensures the safe use of an unverifiable complex controller by using a verified safety controller and verified switching logic. This architecture enables the safe use of high-performance, untrusted, and complex control algorithms without requiring them to be formally verified. The architecture incorporates a supervisory controller and safety controller that will take over control if the unverified logic misbehaves. The supervisory controller should (1) guarantee the system never enters an unsafe state (safety), but (2) use the complex controller the majority of the time (minimize conservatism).

II. Background

- States which do not violate any of the operational constraints are called **admissible** states. Those which violate the constraints are called **inadmissible** states.
- The set of **recoverable** states is a subset of the admissible states, such that if the given safety controller is used from these states, all future states will remain admissible.

III. Control System Monitoring

A. LMI Simplex Architecture

In this approach, system dynamics are approximated by a linear model using the standard control-theoretic approach, where $x' = Ax + Bu$ for state vector x and input u. The operational constraints and saturation limits are expressed as linear constraints in a Linear Matrix Inequality (LMI). These constraints, along with linear dynamics for the system are entered into a convex optimization problem that produces a linear proportional controller gains K and a positive-definite matrix P. The controller produced is a linear-state feedback controller, $u = Kx$, yielding the closed loop dynamics $x' = (A + BK)x$. The matrix P is constructed such that it defines an ellipsoid in the state space where all the constraints are satisfied when $x^T Px < 1$. Since the saturation constraints are included in the system, the control commands are within the actuator limits when the system is inside the ellipsoid. Therefore, the controller gain K defines a safety controller inside of the ellipsoid of states $x^T Px < 1$.

The decision module needs to switch to the safety controller before the system leaves the ellipsoid. Thus, a smaller region is used to define the region where the complex controller is allowed to actuate the system. The buffer is defined by the distance d in Figure 2.

Drawbacks

- The LMI approach is limited to linear dynamic systems
- Sub-Optimal Safety Region - (1) the LMI controller stays within the saturation limits of the actuator, but the saturated safety controller could still recover the system. (2) The switching condition is based on a Lyapunov function that always results in an ellipsoid safety region.

B. Reachability - Hybrid Systems

The dynamics of a control system are represented with a hybrid automaton. A hybrid automaton is a formal model for a system with discrete and continuous behavior, represented as a tuple, $H = (X, L, X_0, I, F, T)$.

- X is the set of continuous states
- L is the set of discrete states
- X_0 is the set of initial states
- I is a set of invariants for the continuous states of a discrete state
- F are the differential equations that represent the continuous system in each discrete state

- T are the guard conditions and reset maps for transitioning between discrete states

Semantically, a hybrid automaton behaves by advancing time according to the differential equations defined by the current discrete state $l \in L$. The guard conditions on the outgoing transitions define when the discrete state can change. The invariants of the discrete state can be used to force transitions by preventing time from elapsing further in the current state. This model allows non-determinism in the discrete behavior. A hybrid automaton can be visually depicted as a finite-state machine with differential equations in each discrete state. The model also allows for non-determinism in the continuous behavior because a single state $x \in X$ defines through its differential equations a set of derivative values for each variable.

The reachability Simplex architecture is not limited to linear systems like the LMI Simplex approach. In addition, the conservatism of the system is tune-able by increasing the accuracy of the reachability calculation. The algorithm can exchange higher accuracy for lower computational performance.

Drawbacks

- Over-approximation error from creating a hybrid model to represent every command for complex controller system
- Representing the result of the computation in an efficient manner. The algorithm can generate in a large non-convex set over several dimensions.
- Reachability techniques incur error depending on the size of the initial set and the reachability time bound

Over-approximation error - The reachability analysis overestimates the set of reachable states. The system is safe, if the reachable set does not violate the safety constraints, because the over-approximation encapsulates the actual reachable set.

IV. Real-Time Reachability Monitor Algorithm

Theorem 1: A safe switching condition for the Simplex architecture is given when, at every control iteration, the complex controller is used if, for some time, (1) $REACH \leq \delta(x,CC) \cup U = \emptyset$, (2) $REACH \leq \alpha(REACH = \delta(x,CC),SC) \cup U = \emptyset$ and (3) $REACH = \alpha(REACH = \delta(x,CC),SC) \subseteq R$.

1) Condition 1 - The Complex Controller (CC) does not intersect with the inadmissible states until time δ.
2) Condition 2 - The Safety Controller (SC) does not intersect with the inadmissible set given the REACH set for the complex controller at time δ until time α
3) Condition 3 - At time α, the safety controller is inside the recoverable region given the REACH set of the complex controller at time δ

The safety condition states that the safety controller is capable of returning the dynamic system to the recoverable region. Therefore, the complex controller is allowed to operate outside of the bounds of the recoverable region.

A. Combine LMI and Reachability Simplex Architectures

A summary of the proposed approach is as follows: when the system is well-inside the ellipsoid that represents the Lyapunov function, we do not need to invoke an extensive reachability analysis. When the system approaches the boundary of the ellipsoid, the run-time reachability analysis is used to allow the system to cross the boundary of the ellipsoid under certain safety conditions. (1) No system constraints are violated when the system leaves the ellipsoid. (2) The state can be guaranteed to be brought back into the ellipsoid.

This approach allows the complex controller to be used in a larger region when compared with the LMI-approach because it can soundly reason about the behavior of the system outside of the ellipsoid. This condition can also be less conservative than the pure reachability approach because the computation starts from a single state x, rather than a potentially large set of inadmissible states. Additionally, it involves reasoning over a finite-time horizon ($\alpha + \delta$), rather than infinite-time reachability.

B. Requirements for Real-Time Reachability Algorithm

- Fast computation for short reachability time calculations
- No dynamic data structures or recursion
- Check for the three conditions from Theorem 1
- Greater result accuracy with more computation time

C. Algorithm Overview

The reachability set of the dynamic system is represented by a hyper-rectangle. The hyper-rectangle is represented with an array of the min-max values of every face. Since the width of the hyper-rectangle cannot be zero for every face, a small ε offset is added to some face of the hyper-rectangle.

Algorithm 1 The real-time, face-lifting reachability algorithm uses a desired reach-time step to tune its run-time

```
1: HyperRect state = initialState
2: while reachTimeRemaining >= 0
3: HyperRect[] nebs = constructNebs(state, reachTimeStep)
4: crossTime = minCrossReachTime(nebs)
5: advanceTime = min(crossTime, reachTimeRemaining)
6: state = advanceState(nebs, advanceRTime)
7: reachTimeRemaining -= advanceTime
8: end while
```

The hyper-rectangle is checked during each iteration of the while loop to ensure that the system does not violate the safety constraints of the system.

1) ConstructNeighborhood function
    a) Create a flat neighborhood for each face of the hyper-rectangle
    b) Calculate the min/max derivative for the min/max flat neighborhood
    c) Construct neighborhoods for each face based on the maximum derivative in each direction
    d) Re-sample derivatives for each neighborhoods

e) Reconstruct the neighborhoods if the neighborhood flips from inward-facing to outward-facing or the width of the neighborhood doubles in size

2) Select the minimum time to leave every neighborhood of the hyper-rectangle
3) Advance the hyper-rectangle state with the minimum reach time and the derivatives of the neighborhoods

Reconstruct Neighborhood Summary

The reconstruction phase of the Construct Neighborhood function ensures that the reach time progress through the face is at least (reachTimeStep / 2). Therefore, the number of iterations of the while loop is bounded by the desired reachTime divided by (reachTimeStep / 2). The reachTimeStep parameter determines the size of the neighborhoods and controls the speed and accuracy of the algorithm.

V. Copilot Version of RTR Monitor

A. Information provided by the monitor specification

- Number of dimensions for the state space
- Number of inputs
- Linear System - $x' = (A+BK)x$ - Matrix A, B, K
- Non-Linear System - (1) A function that returns the derivative in each dimension of the state space and (2) a function that checks the inflection points of the state space
- Input saturation constraints
- A function that returns the minimum and maximum derivatives in an arbitrary hyper-rectangle of the state space
- A function that determines whether a given hyper-rectangle is contained entirely inside the safe region of the system

B. Type Signature of Data Types

- data Interval = Interval {minV :: Float, maxV :: Float}
- data Neighborhood = Neighborhood { delta :: [Interval], width :: [Interval] }
- data RNeighborhood = RNeighborhood { reconstruct :: Bool, community :: Neighborhood }

C. Type Signature of Monitor Functions

- facelift :: [Interval] $->$ Float $->$ [Interval]
- constructNeighborhood :: [Interval] $->$ Neighborhood
- reconstructNeighborhood :: [Interval] $->$ Neighborhood $->$ Neighborhood
- generateNeighborhood :: [Interval] $->$ Int $->$ Bool $->$ [Interval] $->$ [Interval]
- resampleDerivative :: Bool $->$ Float $->$ Float $->$ Bool
- flatNeighborhood :: [Interval] $->$ Int $->$ [Interval] $->$ [Interval] $->$ Neighborhood
- generateFlatNeighborhood :: [Interval] $->$ Int $->$ Bool $->$ [Interval]
- getMinMaxDerivative :: [Interval] $->$ Int $->$ Int $->$ Float $->$ Float $->$ Interval
- generateHyperPoint :: Int $->$ [Interval] $->$ [Float]
- minCrossReachTime :: [Interval] $->$ [Interval] $->$ Float $->$ Float
- advanceBox :: [Interval] $->$ [Interval] $->$ Float $->$ [Interval]
- derivative :: Matrix Float $->$ Matrix Float $->$ Matrix Float $->$ Matrix Float $->$ Matrix Float
- boundInput :: Matrix Float $->$ Matrix Float

VI. Discussion

Haskell has a Data.Matrix package that supports native matrix operations. There is a function, **fromList** in the Data.Matrix package, which will create a Matrix data type from a list. Dynamic systems represent their state space using vectors. The analog to vectors in C is an array. The Copilot language should support reading a C array as a list under a new type of **extern** variable. The list is then converted into a Matrix using the **fromList** function. The matrix is used as an input to the update function of a dynamic system.

A. Things I don't understand

- What is the purpose of the neighborhoods?
- Why do the neighborhoods need to encapsulate the next set of tracked states?
- Why are the derivative not supposed to leave the neighborhood boundaries?

B. Type Signature

extern Matrix :: (RealFloat a) => String $->$ Int $->$ Int $->$ Maybe [a] $->$ Stream (Matrix Float)

- Variable $->$ Name $->$ #Rows $->$ #Cols $->$ Environment $->$ Stream

References

[1] S. Bak, T. T. Johnson, M. Caccamo, and L. Sha, "Real-time reachability for verified simplex design," in *Real-Time Systems Symposium (RTSS), 2014 IEEE*. IEEE, 2014, pp. 138–148.

[2] L. Pike, A. Goodloe, R. Morisset, and S. Niller, "Copilot: a hard real-time runtime monitor," in *Runtime Verification*. Springer, 2010, pp. 345–359.