SMT-based Verification of Safety-Critical Embedded Control Software

Abstract—**A large fraction of bugs discovered in the design flow of embedded control software arises from the interaction of the control software with the plant it controls. Traditional formal analysis approaches using interleaved controller-plant reach-set analysis grossly over-approximate the reachable states and doesn't scale. In this article, we examine a verification approach that considers a control system with the (possibly nonlinear) plant dynamics and mode switches specified along with the actual control software implementation. Given this input, we generate a bounded-time safety verification problem encoded as Satisfiability Modulo Theories (SMT) constraints. We leverage δ-decidability over Reals to achieve scalability while verifying the control software.**

Index Terms—**Sampled Data Systems, δ-Approximation, Control Software Verification**

I. Introduction

Embedded control software (ECS) systems are at the core of safety-critical functionalities present in domains like automotive, healthcare, avionics etc. Safety of such systems rely heavily on the controller implementation and closed loop plant-controller interactions. An isolated verification of the control software can reveal presence of software bugs and computational precision errors. Program analysis based software verification focus on validating properties like absence of overflow, arithmetic exceptions etc through reachability analysis. However, such approaches have the following issues. First of all, any counterexample produced during such verification efforts may not be realizable in a closed-loop setting. A software only verification effort simply ignores the closed loop plant dynamics and properties related to multiple control iterations, which is common for most useful control system properties like performance, stability etc. On the other hand, Hybrid Automata (HA) based approaches which model multi-mode plant dynamics using locations and control updates using transitions are often found introducing significant over-approximation error during reach-set computation while attempting to verify such properties [1].

In summary, the challenge with performing control software verification in the loop with the actual plant dynamics, stem from the following issues.

- Functional equivalence: The actual control software implementation may possibly differ from the mathematical control law, as perceived by the control designer.
- Platform effects: Control law updates may not be happening at the exact sampling instants, but may get delayed due to platform level uncertainties.
- Non-linearity (at plant or controller side): while this may be handled by linearization in case of plants, it is not really the same with non-linear control designs.

For most closed loop applications, controllers are developed using a Model Based Design (MBD) strategy. In MBD paradigm, the plant and the controller models are represented as continuous and discrete blocks consisting of continuous plant dynamics and discrete switching logic respectively, arranged in a closed loop formation. This approach is popular as it enables design, testing and verification to be performed in single platform (eg. Stateflow / Simulink). For verification purpose, such models are translated to abstract HA, which is commonly analysed using formal reachability analysis tools. Later, in the system development life cycle, as the developer finalises the mathematical model after several stages of corrections and revisions as needed, the model is ready for implementation. For implementation of this controller model into embedded board (to run in closed loop with plant), Code Generation tools (eg. Embedded Coder/Simulink Coder) present in the MBD platforms are used. These code translations are mostly unverified and can introduce subtle bugs [2], [3]. Moreover as said earlier, common implementation platforms introduce timing uncertainties during closed loop execution of such software tasks.

The execution timings of embedded control task instances are often non-ideal due to effects like sampling-jitter, sensing delay, interference from other control tasks sharing a common execution platform etc. These issues introduce variability in the response time of the control task which further manifest in terms of control execution drops which influence the stability and the performance of the closed-loop system. However, such issues are usually ignored in verification strategies for control software implementations. In effect, although the high level abstract model is verified, we need to verify if the actual software controller implementation satisfies the safety and performance specifications in closed loop over a finite number of consecutive control iterations, (as is required by the relevant property). There are tools like Sahvy [4], that performs safety verification of closed loop systems, where the plant is periodically sampled and actuated at discrete intervals by the control program. Sahvy uses SMT solver Z3 to generate assertions over the software state and FLOW* for obtaining the reachable set of the plant but does not consider the effect of timing uncertainties and disturbances. In [5] Duggirala et al. have performed a closed-loop analysis of control programs performed on a Real-time operating system, but it is limited

only to linear systems.

However, verification of a controller implementation with the plant model, which is non-linear in general, is challenging. Simulation-driven verification tools [6], [7] are reported to scale well for industrial benchmarks. They use a falsification based approach with the robustness semantics of a given property as a cost function. There are tools with numerical simulations for the plant model and the symbolic execution for the control program [8]. For the purpose of formal verification of non-linear systems, one needs to consider satisfiability problems over Reals, which is undecidable in general. However, with a given arbitrary precision, $\delta > 0$, these problems become decidable [9] and can be solved using δ-approximation SMT solvers like dReal [10]. dReach [11] and iSAT-ODE [12] are two such tools that perform bounded model checking of hybrid systems with the help of SMT solvers dReal and iSAT respectively. Both tools can handle expressive models with complex non-linear dynamics and reports verification of industrial case-studies.

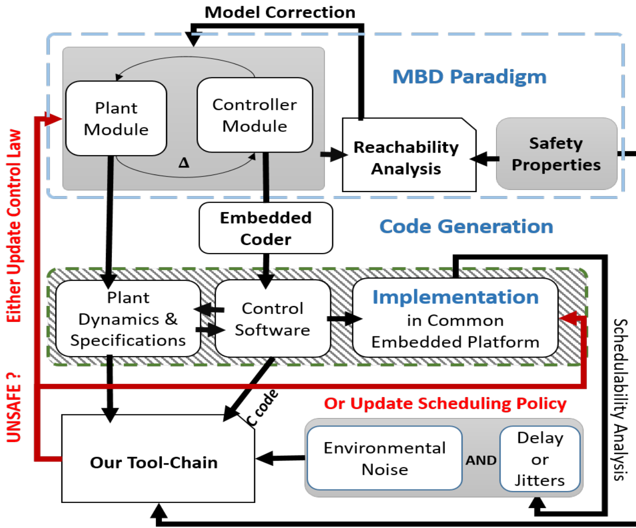

Fig. 1. Verification in MBD (*blue dashed box*) and Implementation (*patterned box*) phase (*grey boxes denote inputs to our tool*).

In our work, we leverage the idea of δ-decidability with an additional primitive, we create an SMT based representation not only of the plant dynamics, but also for the control software implementation. We consider an implemented control software operating in iterations, at a fixed sampling time period Δ, in closed loop with a continuous plant. Our tool-chain (refer to Fig. 1) proposes to verify the actual implementation of a nonlinear control system i.e. C code executing in loop with the plant dynamics under various disturbances, delays and jitters.

In summary, following are our major contributions.

1. Since our tool-chain considers an actual C implementation of the controller, our tool-chain can potentially verify a significant class of real life embedded control software.

2. Our tool-chain handles nonlinearity of a closed loop control implementation by generating an SMT based encoding of the closed loop and then leveraging the theory of δ-decidability over Real numbers as supported by the dReal solver. Leveraging δ-decidability provides us a tuning parameter for choosing the suitable level of precision, a feature which we exploit to handle large state spaces while preserving soundness of result.

3. Our tool incorporates semantic support for capturing timing uncertainties like delay, jitters, and value based uncertainties like sensor noise in order to check their effect on the performance and safety of the closed loop. By considering these as separate inputs, we are able to skip introducing them in the top level HA model and retain the HA based plant dynamics specifications as intuitive representations of the original mathematical models. In the following section, we present our proposed methodology. To demonstrate the correctness of our approach, we provide analysis reports using our tool on various benchmarks of closed loop control systems.

II. Framework for ECS Verification

The plant dynamics of a single mode discrete control system,

$$\dot{x} = f(x, u, w) \tag{1}$$

where, flow function $f : \mathcal{X} \times \mathcal{U} \times \mathcal{W} \to \mathcal{X},\ x \in \mathcal{X} \subseteq \mathbb{R}^{d_x}$ is system state vector, $u \in \mathcal{U} \subseteq \mathbb{R}^{d_u}$ is control input vector, $w \in \mathcal{W} \subseteq \mathbb{R}^{d_w}$ is input disturbance vector and d_x, d_u, d_w denote the dimension of state, control input and disturbance vectors respectively. The control input u remains constant during the sampling period (Δ). Due to the presence of delay/jitters, the actual time T_k between k^{th} and $(k+1)^{th}$ control updates becomes $T_k = t_{k+1} - t_k = \Delta + \epsilon_k$, where the sampling jitter $\epsilon_k \in [0, \epsilon]$ in the k-th execution period. We consider the upper bounds on sampling jitter and quantization error for certain variable as $\epsilon \geq 0$ and $\delta_{var} > 0$ respectively.

Tool Input Specifications: The input specification for our tool as shown in Fig. 2 is described next.

(1) The *Plant model* file contains the plant dynamics specification. Our modeling method follows the specification format of [13] (called HASLAC) for capturing plants having multi-mode dynamics, with Ordinary Differential Equations (ODEs) describing the dynamics in each mode. In our modeling formalism, we consider that the plant can have multiple flows, $f_1, f_2, \cdots$, one for each mode. This implies that the plant can switch modes by itself based on guard conditions defined over $\mathcal{X}$ as well as if dictated by the control software.

(2) The *Control software* is a C-program file which is the controller implementation generated by Matlab Embedded Coder toolbox. In earlier stages of MBD, high level tools like Simulink/Stateflow are used to generate a plant-controller model. Once verified to work properly in simulation this controller code is then generated using inbuilt translators (e.g. Embedded Coder). In our tool-chain, we use a standard input format of the controller program. It helps omit any possibilities of datatype mismatch. Plant states and control commands between plant and controller program are exchanged through two global data-structures respectively: `INPUT_VAL`(x) and `RETURN_VAL`(u), a convention used in Embedded Coder with x and u denoting plant state and the control actions as usual.

(3) The *Configuration file* specifies two types of input parameters (these can also be input via command line). *A. System Properties*: This part contains **(i)** minimum and maximum

value bounds for system variables, **(ii)** goal property i.e., negation of the safety property that is to be verified.

B. Uncertainties & Disturbances: This part contains fields for the following entries. **(i)** *Sampling jitter and Response time jitter*: are timing uncertainty for the sensor data to be read by the software and latency for the software to compute the actuator parameters respectively. We model both of these collectively to be input as ϵ to represent timing uncertainty caused in every control loop iteration. **(ii)** *Noise*: specifies environmental disturbances (w) affecting the sensor data values, **(iii)** *Quantization error*: captures the deviations that may occur due to precision errors (δ_{var}) in sensed or actuated values due to fixed-point controller implementation. These uncertainties and disturbances can be input being specific to variables. After parsing we accordingly map them to the equations of corresponding variables.

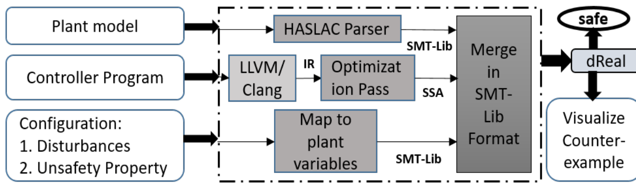

Fig. 2. Tool-flow for Verification of Embedded Control Software.

Tool Design: Figure 2 gives an overview of the different steps, executed by the tool. The overall functionalities are formalized in Algorithm 1 and step wise described below.

Algorithm 1 Algorithmic representation of Tool-Flow

Require: Plant Dynamics in HASLAC HA, Controller Program in C $Code$, Safety property $\varphi_{safety}(x)$, Jitter Values ϵ, Disturbances w, Quantization error δ, Initial range for plant and controller $Init = \langle x_{init}, u_{init}, w_{init} \rangle$, Unrolling bound N
Ensure: δ-sat or $unsat$
1: $\langle \mathbf{f}(x, u, w), \Delta \rangle \leftarrow$ HASLACPARSER(HA) ▷ parse plant dynamics
2: $CP(u, x) \leftarrow$ CONTROLLERPARSER($Code$) ▷ parse C program
3: $\varphi \leftarrow$ ENCODESMT($\mathbf{f}(x, u, w), \Delta, CP(u, x), \epsilon, w, \delta, Init, \varphi_{safety}(x)$) ▷ Encodes closed-loop system with the delays, noise and jitters into SMT over Reals
4: CALLDREAL(φ) ▷ Calls dReal Solver to verify the encoded SMT formula
5: **if** φ is δ-sat **then**
6: **return** Counter-Example to rectify controller/scheduling policy.
7: **else**
8: **return** $unsat$
9: **function** ENCODESMT($\mathbf{f}(x, u, w), \Delta, CP(u, x), \epsilon, w, \delta, Init, \varphi_{safety}(x)$)
10: $\varphi \leftarrow null$
11: $x_0^0 \leftarrow x_{init}$, $u_0 \leftarrow u_{init}$, $w_0 \leftarrow w_{init}$, $t_0 \leftarrow 0$ ▷ initializing variables
12: $\varphi \leftarrow \varphi \wedge Init(x_0^0, u_0, w_0)$ ▷ $Init$ is the clause against initialisation of variables
13: **for** $k \in [0, N]$ **do**
14: $\epsilon_k \leftarrow nondet([0, \epsilon])$ ▷ non-deterministic sampling jitter $\epsilon_k \in [0, \epsilon]$
15: $T_k \leftarrow \Delta + \epsilon_k$ ▷ starting iteration ϵ_0 is $release\ time$
16: $w_k \leftarrow nondet([-w, w])$ ▷ non-deterministic noise $w_k \in [-w, w]$
17: $\mathcal{C}_1 : x_k^t \leftarrow \int_{t_k}^{t_k+T_k} \mathbf{f}(x_k^0, u_k, w_k)$ ▷ state progression with disturbances
18: $\delta_{u_k} \leftarrow nondet([-\delta_u, \delta_u])$ ▷ quantization error while actuation
19: $\delta_{x_k} \leftarrow nondet([-\delta_x, \delta_x])$ ▷ quantization error while state estimation
20: $x_k \leftarrow x_k^{t=\Delta} + \delta_{x_k}$
21: $\mathcal{C}_2 : u_{k+1} \leftarrow CP(x_k, u_k) + \delta_{u_k}$ ▷ Control input calculation
22: $x_{k+1}^0 \leftarrow x_k^t$ ▷ State updation
23: $t_{k+1} \leftarrow t_k + T_k$
24: $\varphi \leftarrow \varphi \wedge \mathcal{C}_1 \wedge \mathcal{C}_2$
25: $\varphi \leftarrow \varphi \wedge \neg\varphi_{safety}(x_N^t)$ ▷ safety property check
26: **return** φ

1. ***Model Transformation*****:** The original multi-mode hybrid automaton is transformed into a semantically equivalent single mode representation at first. The multi-mode hybrid automata input using HASLAC can contain multiple flow equations $f_1 \cdots f_k$ from multiple modes. In the transformation step, we replicate each state variable k number of times, in order to model the evolution of the single mode system using one possible flow at a time. We represent this collection of state flow equations using a flow equation vector $\mathbf{f} = [f_1, f_2 \cdots f_k]$. To enable this switching, we introduce *auxiliary* control variables, using which we can mimic the mode switching logic and switch between different flow equations observing different plant states in single mode.

2. ***Parsing Plant Dynamics*****:** The transformed plant model is parsed using the HASLAC parser as a single mode automaton object (Line 1). This hybrid automaton model [14] contains all the flow equations (sets of ODEs) following which the plant variables evolve over time, guards and invariant conditions (eg. sampling interval, i.e. how frequently the plant should interact with controller to update the state).

3. ***Parsing Controller Program*****:** The control program next is parsed using the *Clang/LLVM* library. The program then is translated to LLVM bitcode and the bitcode IR is then converted into *Single Static Assignment (SSA)* form for tracking the evolution of controller variable with time progress. Using appropriate LLVM code this representation can be converted into an SMT encoding. This SMT encoding essentially represents a functionally equivalent logical representation of the controller program, expressed as $CP()$ (in Line 2).

4. ***SMT Formulation and Verification***: The `EncodeSMT()` function in Algorithm 1 automatically creates the SMT formulation that captures the system progression. It generates an assertion φ that contains SMT formula of closed loop system evolution via plant-controller communication for N iterations in presence of non-deterministic noise and jitters. The closed-loop execution starts with an initial set (x_0, u_0, w_0), i.e. the initial values of x, u, w respectively (Line 11). A continuous flow from x_k^0 to x_k^t in time T_k is governed by Eqn.(1) in every iteration i.e. the plant progresses following, $x_k^t = \int_{t_k}^{t_k+T_k} \mathbf{f}(x_k^0, u_k, w_k)$ (Line 17). Here w_k is the non-deterministic process noise within the user input range (w, see Line 16). The interval T_i is affected by a non-deterministic sampling jitter ϵ_k within user specified range ($[0, \epsilon]$) (Line 14-15). In each sampling step, the parsed and simplified control program $CP()$ computes the next control output u_{k+1} using last updated plant state x_k^t, i.e. $u_{k+1} = CP(u_k, x_k^t)$ (Lines 20-21). A quantization error value tuple (δ_x, δ_u) is non deterministically chosen from user input range and added during actuation (Lines 18, 21) and state estimation (Lines 19-20). All the uncertainty and disturbance parameters, like sampling jitter, noise, quantization errors and the unroll bound N are provided by the user in the configuration file. In each iteration of the loop (Line 24), SMT clauses are created capturing the possible trajectories in each iteration and added to the formula under construction. Our goal is to check if the reachable domain of final state following the state progression as captured by the SMT is safe. At the end of N iterations of the `for` loop, we have an overall forward reachability formula that is put in conjunction with the negation of desired safety property $\neg\varphi_{safety}$ (Line 25) to give the final formula φ, returned by

EncodeSMT(). In symbolic form, the overall formula for N iterations becomes, $\varphi = Init(x_0^0, u_0, w_0) \wedge \bigwedge_{i=0}^{N-1} \Big[(x_i^t = \int_{t_i}^{t_i+T_i} \mathbf{f}(x_i^0, u_i, w_i)) \wedge (u_{i+1} = CP(u_i, x_i + \delta_{x_k}) + \delta_{u_k}) \Big]$ $\wedge \neg\varphi_{safety}(x_N^t)$, with clauses for non-deterministic choice abstracted for brevity. We use SMT-LIB version 2.0 to formulate this assertion, as it has extensions to declare systems of ODEs [9]. Our tool finally generates a file containing the SMT encoding of assertion φ following above equation in prefix notation (Line 3). The returned SMT formula, capturing the plant dynamics, timing and quantization effects and the actual semantics of the control software, is then passed to the dReal solver (Line 4) for satisfiability check. On getting δ-SAT decision from dReal, the tool reports that a counterexample exists in presence of a δ perturbation bound on the variables, that takes the system to an unsafe situation within N unrolling (Line 6). If the tool reports UNSAT, we have a guarantee that the implemented closed loop control software is safe for the assumed condition bounds (Line 8). In general, counter example traces provide the system designers useful information about possible implementation solutions, e.g. updating the mathematical control law or changing scheduling policy, task mapping etc in the embedded platform (refer Fig 1).

III. Experimental Results

We evaluate our approach on a set of well-known safety-critical CPS benchmarks. The control program is either an abstract version of the actual program or a piece of generated C-code from the Embedded Coder toolbox. This code is annotated to satisfy the requirement for our tool-interface. We perform our experiments on a four-core Intel Xeon(R) 3.50 GHz CPU E5-2637 v4 with 255 GB of RAM.

TABLE I
Summary of the Benchmark Designs

Benchmark	$\mathbf{d_x}$	$\mathbf{\Delta}$[sec]	k	LOC	δ	RT[sec]	Result
Thermostat	2	0.2	5	72	0.001	60.814	δ-SAT
DC Motor	3	0.02	50	43	0.01	96.67	UNSAT
Yaw-Damper	6	0.05	200	21	0.001	51.95	UNSAT
Powertrain	9	0.01	50	177	0.001	94.34	δ-SAT
Lunar Lander	6	0.128	80	30	0.01	142.36	UNSAT
EMB	4	0.001	23	39	0.001	113.56	UNSAT
ACC	1	0.02	6	20	0.01	8915	δ-SAT

$\mathbf{d_x}$ =dimension of system, $\mathbf{\Delta}$ = Sampling Period, $\mathbf{k}$ = the number of iterations, $\mathbf{LOC}$ = Line of Code, δ = Precision and $\mathbf{RT}$ =Run Time of the tool-chain

In the *thermostat* model [8], we observe that the temperature drops below the specified safe value ($52°F$) when there is a sampling jitter of 0.1 second. The *DC Motor* system has two state-variables, armature current(i) and angular velocity($\dot{\theta}$). The verification task is to check whether a forbidden region defined as $i \in [1.0, 1.2] \wedge \dot{\theta} \in [10, 11]$ is reachable by the PI control logic. This set is chosen because it is observed that this combination is hard to reach with random simulations [8]. We have also successfully verified the correctness of the controller in 1) *Yaw-damper for a 747 aircraft* [1], with 200 closed-loop iterations, 2) *Powertrain* benchmark [15] (*non-linear* controller) with 50 closed-loop iterations. The *Yaw-damper* and *Powertrain* benchmarks have been analyzed for steady-state spiral-mode and normal-mode respectively. The goal of the *non-linear* controller in *Powertrain* benchmark [15] is to keep the air-to-fuel ratio close to an 'ideal' stoichiometric ratio. In similar veins, we have also performed successful verification of *Descent Guidance of a Lunar Lander* [16], *Adaptive Cruise Controller (ACC)* [17] and *Electro-Mechanic Braking System (EMB)* [18]. Details about target properties and initial configurations are available in cited literature and in the provided web link [19].

IV. Conclusion

We present a tool framework to verify the safety property of closed-loop systems under the influence of timing uncertainties and environmental disturbances. We demonstrate bounded-time verification results on benchmarks of standard control systems. In the tool, the designer can directly input the controller program along with the plant dynamics and observe the impact of data and timing discrepancies on the safety requirements of the actual software implementation. We plan to extend the tool in terms of scalability and handling of complex non-linear systems in future using novel techniques for approximate reachability analysis.

References

[1] S. Bak *et al.*, "Periodically-scheduled controller analysis using hybrid systems reachability and continuization," in *RTSS*. IEEE Computer Society, 2015, p. 195–205.

[2] J. Park, *et al.*, "LCV: a verification tool for linear controller software," in *TACAS*. Springer, 2019, pp. 213–225.

[3] J. Park *et al.*, "Scalable verification of linear controller software," in *TACAS*. Springer-Verlag, 2016, p. 662–679.

[4] G. Simko *et al.*, "A bounded model checking tool for periodic sample-hold systems," in *HSCC*. ACM, 2014.

[5] P. S. Duggirala *et al.*, "Analyzing real time linear control systems using software verification," in *RTSS*. IEEE, 2015.

[6] Y. Annpureddy *et al.*, "S-TALIRO: A tool for temporal logic falsification for hybrid systems," in *TACAS*. Springer, 2011.

[7] A. Donzé, "Breach, a toolbox for verification and parameter synthesis of hybrid systems," in *CAV*. Springer, 2010.

[8] Zutshi *et al.*, "Symbolic-numeric reachability analysis of closed-loop control software," in *HSCC*. ACM, 2016.

[9] S. Gao *et al.*, "Satisfiability modulo ODEs," in *2013 Formal Methods in Computer-Aided Design*. IEEE, 2013, pp. 105–112.

[10] S. Gao, J. Avigad *et al.*, "δ-complete decision procedures for satisfiability over the reals," in *IJCAR*. Springer, 2012.

[11] S. Kong *et al.*, "dReach: δ-reachability analysis for hybrid systems," in *TACAS*. Springer, 2015.

[12] A. Eggers *et al.*, "Improving SAT modulo ODE for hybrid systems analysis by combining different enclosure methods," in *SEFM*. Springer, 2011.

[13] A. A. B. da Costa *et al.*, "ForFET: A formal feature evaluation tool for hybrid systems," in *ATVA*. Springer, 2017, pp. 437–445.

[14] T. A. Henzinger, "The theory of hybrid automata," in *Verification of digital and hybrid systems*. Springer, 2000, pp. 265–292.

[15] X. Jin *et al.*, "Powertrain control verification benchmark," in *HSCC*. ACM, 2014.

[16] H. Zhao *et al.*, "Formal verification of a descent guidance control program of a lunar lander," in *FM 2014*. Springer, 2014.

[17] "Cruise Control: System Modeling," http://ctms.engin.umich.edu/CTMS/index.php?example=CruiseControl§ion=SystemModeling.

[18] T. Strathmann *et al.*, "Verifying properties of an electro-mechanical braking system," in *ARCH14-15*. EasyChair, 2015, pp. 49–56.

[19] "SMT benchmarks with configurations," https://sites.google.com/view/benchmarkssafeemc/home.