The Design of Intelligent Temperature Control System of Smart House with MARS

Yihao Yin[1,2,3], Hao Wu[2,3], Shuling Wang[3,4(✉)], Xiong Xu[3,4], Fanjiang Xu[3,4], and Naijun Zhan[4,5]

[1] Hangzhou Institute for Advanced Study (HIAS), UCAS, Beijing, China
[2] Key Laboratory of System Software, ISCAS, Beijing, China
[3] University of Chinese Academy of Sciences, Beijing, China
wangsl@ios.ac.cn
[4] National Key Laboratory of Space Integrated Information System, ISCAS, Beijing, China
[5] School of Computer Science, Peking University, Beijing, China

Abstract. MARS is a toolchain, supporting model-based design of cyber-physical systems (CPS), which integrates informal and formal design. With MARS, a system under development can be graphically modeled by the combination of AADL and Simulink/Stateflow, then the simulation of the graphical model can be conducted. Furthermore, the graphical model can be automatically transformed to Hybrid Communicating Sequential Processes (HCSP) for formal verification with HHL-Prover. Finally, ANSI-C code or SystemC code can be generated from the verified HCSP formal model with the guarantee of correctness. As a case study of CPS, in this paper, we apply MARS to design an intelligent temperature control system, including its modeling, simulation, verification and code generation. This case study demonstrates the advantages of the design of CPS with MARS, including the integration of modeling, simulation, verification and code generation; the integration of informal and formal design, thus providing balance between efficiency and rigidity.

Keywords: Simulink/Stateflow · Model-based design · HCSP · Code Generation · Verification

1 Introduction

The applications of embedded systems (nowadays called Cyber-Physical Systems (CPS)) are extremely broad encompassing nearly every aspect of modern life, especially in many safety-critical areas such as autonomous driving, medical devices, aerospace and so on. For such systems, any mistake of them may

T. Bourke et al. (Eds.): SETTA 2024, LNCS 15469, pp. 217–235, 2025.
https://doi.org/10.1007/978-981-96-0602-3_12

result in catastrophic consequences. However, complex CPS involve closely coupling of discrete control, continuous plants and communications, thus how to efficiently design reliable CPS is very challenging. Both industrial and academic communities have paid increasing attention to design safe CPS, which can be categorized into simulation-based, formal methods based, and their combination. Simulation-based approaches are advocated by industry, such as Simulink/Stateflow (S/S) [15] and AADL [5]. S/S has become a de facto model-based design tool in embedded industry, but it is insufficient for the design of safety-critical CPS because of the inherent incompleteness of simulation. AADL provides architecture modeling and analysis of CPS by simulation, and furthermore supports the automated code generation from AADL models to C code. However, it cannot support modeling continuous physical processes as well as their combination with software. Formal methods based approaches are advocated by academic community, which can be further classified into model-checking based and theorem proving based. In model-checking based approaches, a CPS is modeled as a hybrid automaton [1,8], and verification is done by computing reachable states [4,6,10]. In theorem proving based approaches, a CPS is modeled by a compositional modeling language, and verification is conducted through theorem proving, e.g. differential dynamic logic (dL) [18,31]. SCADE [2] tried to combine formal and informal, but failed to bridge the gap between informal graphical models and formal algorithm models.

In order to bridge the gap between informal and formal model-based design for CPS, in our previous work, we developed a toolchain called MARS [30], supporting modeling, analysis, verification, and code generation for CPS. MARS starts to design a graphical model for the system to be developed using the combination of AADL and S/S, by considering the functionality, physicality and architecture of the system in a unified framework [27]. Then, formal analysis and verification of the combined graphical model can be conducted via the translation of AADL and S/S into Hybrid CSP (HCSP), an extension of CSP for formally modeling hybrid systems [28]. The HCSP models can be simulated using the HCSP simulator. Additionally, to complement incomplete simulation, they can be verified using HHLProver (Hybrid Hoare Logic prover) implemented in Isabelle/HOL [23], as well as a more automated HHLPy prover [20]. Finally, implementations in SystemC or C can be automatically generated from verified HCSP models [24,29]. The transformation from the combined AADL and S/S to HCSP, and the one from HCSP to SystemC or ANSI-C, are both guaranteed to be correct [24,26]. MARS provides model-based design of safety-critical CPS by allowing switching between formal and informal seamlessly, depending on the efficiency, cost and rigidity.

In this paper, we apply MARS to the design of an intelligent temperature control system (ITCS), including its modeling, simulation, verification, and code generation. Specifically, the graphical model of the system is constructed using S/S, and then it is translated into an HCSP model, based on which simulation and verification are performed. From the verified HCSP model, we continue to generate ANSI-C code, which is guaranteed to be reliable given the correctness of the translation proved. The goal of this paper is to demonstrate the entire process of the model-based design approach, by applying MARS for the modeling, simulation, verification, and code generation of the case study, thereby validating the applicability of MARS for the design of complex embedded systems.

Paper Organization. The rest of the paper is organized as follows. Section 2 introduces some preliminary knowledge of this paper. Section 3 introduces the ITCS case study, and Sect. 4 presents the modeling, simulation, verification, and code generation of ITCS using MARS. Finally Sect. 6 concludes the paper.

2 Background

In this section, we introduce some preliminary knowledge for this paper, including Simulink, HCSP, and the MARS toolchain.

2.1 Simulink

Simulink [15] is a graphical environment for model-based design of dynamical systems, supporting description of both discrete-time and continuous-time behavior. A Simulink model contains a set of blocks, subsystems, and wires, where blocks and subsystems cooperate by exchanging data flows through connected wires. Wires can be considered as variables holding these data values. As basic units for building Simulink models, each block is defined with input and output ports, and methods that define how outputs and internal states are changed. Blocks can be grouped into subsystems to establish hierarchical diagrams. To ease modeling, Simulink provides an extensive library of pre-defined blocks and subsystems for building and managing diagrams, and also a rich set of fixed-step and variable-step solvers for analyzing dynamical systems through simulation.

2.2 HCSP

Hybrid CSP (HCSP) [7] is a formal language for describing HSs, which is an extension of CSP by introducing ODEs (ordinary differential equations) for modeling continuous evolution. HCSP includes common constructs such as assignment, internal choice, sequential composition and conditional statement. Besides, it includes more constructs explained as follows:

- Input $ch?x$ receives a value along the channel ch and assigns it to variable x. Output $ch!e$ sends the value of e along ch.
- Repetition c^* executes c for a nondeterministic finite number of times.
- Continuous evolution $\langle \dot{\boldsymbol{x}} = \boldsymbol{e} \& B \rangle$ evolves continuously according to the differential equation $\dot{\boldsymbol{x}} = \boldsymbol{e}$ as long as the *domain* B holds, and terminates whenever B becomes false. Communication interruption $\langle \dot{\boldsymbol{x}} = \boldsymbol{e} \& B \rangle \trianglerighteq [\!]_{i \in I} (ch_i * \rightarrow c_i)$ behaves like $\langle \dot{\boldsymbol{x}} = \boldsymbol{e} \& B \rangle$, except it is preempted as soon as one of the communication events $ch_i *$ takes place, and then is followed by the corresponding c_i.
- Parallel composition $pc_1 \|_{cs} pc_2$ behaves as pc_1 and pc_2 run independently except that all communications along the set of common channels cs between pc_1 and pc_2 are synchronized.

MARS enriches HCSP with constructs on modularity, including **module** for encapsulating a sequential process and **system** for parallel composition of modules.

2.3 MARS

The architecture of the toolchain MARS [30] is shown in Fig. 1, where AADL and S/S indicate graphical models that serve as input to the toolchain; HCSP indicates formal models, with their simulation and verification tools; and SystemC and ANSI C indicate generated code. To design a safety-critical system using MARS, users can choose to build graphical models from the top layer and do analysis through simulation by transforming the combined model to C, or build the formal HCSP models directly, which also have a simulator implemented. Formal verification of HCSP models is done by HHL Prover, which includes the interactive verifier implemented in Isabelle [31], the automatic verifier HHLPy for verifying HCSP sequential subset [20], and differential invariant generator for reasoning about ODEs [12]. Finally, implementations in SystemC [29] or ANSI-C [24] are automatically generated from the verified HCSP processes. Both the transformation from graphical models to HCSP models and the one from HCSP to SystemC or ANSI-C, can be done automatically and furthermore are guaranteed to be correct by proving the consistency between the models in different layers based on their formal semantics [24,28]. MARS supports the transformation of subsets of AADL and S/S, which include the main features of CPS including discrete-time control, continuous evolution, event-based control, *etc.*. Our approach allows model-based design of safety-critical CPS based on graphical and formal models and proven-correct translation procedures.

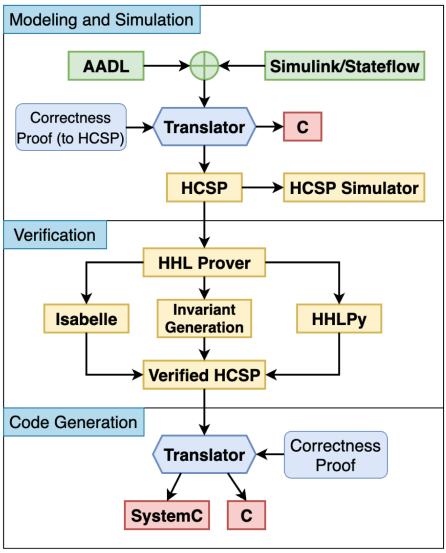


Fig. 1. The Architecture of MARS

3 An Intelligent Temperature Control System (ITCS)

The case study of ITCS is taken from the official website of S/S [15], as shown in Fig. 2. The system is modelled as an S/S diagram, assembled from a combination of continuous blocks, discrete blocks and subsystems, mimicking a real-world scenario wherein the indoor temperature is regulated by automatically toggling the heater on and off in response to changes in outdoor temperature. In this section, we introduce the S/S model of the case study from the overall top structure and the encapsulated subsystems respectively.

3.1 The Overall of ITCS

The system receives inputs from the left two constant blocks, which set the average outdoor temperature to 50 °F and the house temperature to 70 °F respectively. The control system is designed to maintain the indoor temperature at approximately 70 °F, with allowance of given up and down fluctuations. The system uses a sine function to represent the daily outdoor temperature variation and superimposes it over `OutTemp` (i.e. 50) to model the changing outdoor temperature. The F2C block converts the temperature from Fahrenheit to Celsius, then the converted `Tout` is sent to the House subsystem as one input. Concurrently, the constant house temperature 70 °F is also converted via F2C block and then the difference of it with the actual house temperature is calculated (i.e. `Terr`), to be input of the Thermostat subsystem, which determines whether the heater should be activated or not. The Thermostat block then transmits its judgment (i.e. `blowercmd`) as input to the Heater subsystem. The Heater subsystem

receives the actual house temperature as another input and calculates the heat flow, i.e. `HeaterOut`. The heat flow is sent to the House subsystem as another input, and moreover, it is integrated via an integrator block and then multiplied with a constant via a gain block, to obtain the final cost, i.e. `HeatCost`. At the same time, the system calculates the real-time indoor temperature `HouseTemp` through the House subsystem. The final output graph is a line chart composed of the indoor and outdoor temperatures in the form of Fahrenheit degree, and the cost of the heater. The subsystems Thermostat, House and Heater will be explained subsequently.

3.2 The Subsystems

The model of ITCS consists of three subsystems: Thermostat, House and Heater, explained in the following parts.

Thermostat Subsystem. The Thermostat subsystem contains only one Relay block, as shown in Fig. 3a. It maintains an internal state that records the status of the switch. When the input signal exceeds a certain threshold (rise threshold, 215/9 °C here), the switch closes and the output is 0, turning off the heater; when the input signal is below another threshold (fall threshold, 165/9 °C), the switch opens and the output is 1, turning on the heater; otherwise, when the input signal is between 165/9 °C and 215/9 °C, the switch is not changed and the output keeps the value of the switch state. Due to the control of Thermostat subsystem, the indoor temperature can be maintained within a certain range.

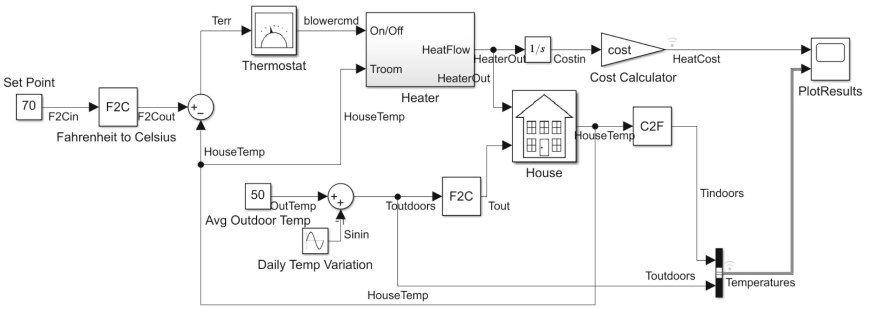


Fig. 2. The S/S Model of ITCS [15]

Heater Subsystem. The Heater subsystem implements the heater as shown in Fig. 3b. The input `On/Off` receives the output command of Thermostat subsystem (1 or 0), `Troom` is the actual house temperature from the House subsystem, and `Theater` is the temperature of the hot air from the heater, which is set to

constant 50 °C here. When the heater is on, the output `HeatFlow` is calculated by the following equation:

$$(\frac{dQ}{dt})_{heater} = (T_{heater} - T_{room}) * Mdot * C$$

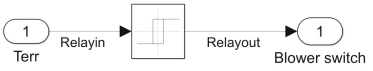
where,$(\frac{dQ}{dt})_{heater}$ represents the heat flow from the heater into the room, $Mdot$ the air mass flow rate through the heater (kg/hr), C the heat capacity of air at constant pressure, and T_{heater}, T_{room} correspond to `Theater` and `Troom` respectively. The output `HeaterFlow` will serve as an input to both the integrator block and the House subsystem, as shown in Fig. 2.

House Subsystem. The House subsystem controls the indoor temperature of the house, as depicted in Fig. 3c. The input `In` receives the heat flow generated by the Heater, and `Tout` inputs the outdoor temperature. It calculates the final indoor temperature based on the inputs using the following equations:

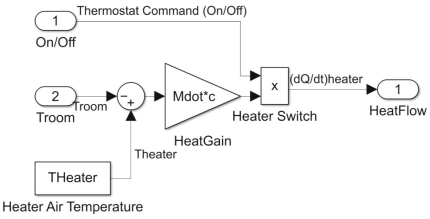
$$(\frac{dQ}{dt})_{losses} = \frac{T_{room} - T_{outdoor}}{R_{eq}}$$

$$\dot{T}_{room} = \frac{1}{M * C} * ((\frac{dQ}{dt})_{heater} - (\frac{dQ}{dt})_{losses})$$

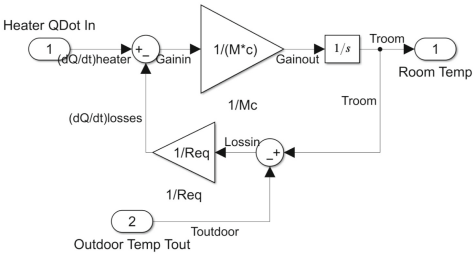
where, $T_{outdoor}$ represents the outdoor temperature `Tout`, T_{room} and C defined as above, R_{eq} the equivalent thermal resistance of the house, M the mass of air inside the house. As shown in the equations, $(\frac{dQ}{dt})_{losses}$ represents the loss rate of the heat in the environment, which is determined by the difference between the actual room temperature and the outdoor temperature, divided by the house thermal resistance. $\dot{T}_{room}$, the final gained heat rate of the room, also the derivative of the room temperature with respect to time, is the difference between the heat flow rate and the loss rate, divided by $M * C$. A loop is formed as the room temperature is also taken as an input of the Lossin block to calculate the loss rate of the heat. In S/S, the diagrams with algebraic loops are considered invalid, while the blocks which maintain internal states such as Integrator or Unit Delay blocks can break the loop. So with existence of the integrator block for `Troom` in House subsystem, the loop in this subsystem and also the main loop in the top diagram (due to the backward transition from House to previous parts) are both valid. Notice that different from $\dot{T}_{room}$, both $(\frac{dQ}{dt})_{losses}$ and $(\frac{dQ}{dt})_{heater}$ are named in the form of differential equations for ease of understanding their meanings and the relations between each other, without actual differential operations.



(a) Thermostat Subsystem



(b) Heater Subsystem



(c) House Subsystem

Fig. 3. The Subsystems

4 Formal Design of ITCS

In this section, we show how to conduct formal design of ITCS starting from the built graphical model, including constructing its formal model represented by HCSP, its simulation based on the HCSP model, verification of the HCSP model, and code generation from the verified HCSP model.

4.1 Translation to HCSP Model

We first apply the toolchain MARS to transform the S/S model of ITCS to HCSP formal model. Taking the S/S model presented in Fig. 2 in .xml format as input, the tool generates the HCSP model as shown in Fig. 4, which is to be served as the foundation for subsequent simulation and verification of ITCS. The generated HCSP model is a system from the overall structure, which contains one module transformed from the S/S model.

Before introducing the HCSP model, we briefly explain the strategy of MARS for transforming a S/S model. It first determines the sample times of all blocks, including the ones inside subsystems, based on which each block is classified as either discrete or continuous; then separates the whole diagram into discrete and continuous parts; finally, transforms the discrete and continuous parts individually first and then put them together in correct execution order to form the whole HCSP model of the S/S diagram. The complexity of the transformation is $O(n^2)$ where n denotes the number of blocks in source models due to the sorting

procedure in correct execution order. The generated HCSP model for a given S/S diagram $\mathcal{D}$ has the following structure:

$$HCSP(\mathcal{D}) \mathrel{\widehat{=}} \mathsf{Output}; \mathsf{Init}; \big(\mathsf{Discrete}; \langle \dot{t} = 1, \dot{\mathbf{y}} = \Gamma(\mathbf{x}) \& t < period \rangle; \mathsf{TimeUpdate}; \big)^* \quad (1)$$

It starts from Output, which is a sequence of assignments to the outputs of $\mathcal{D}$ by their respective values, followed by the initialisation of variables and then a repetition process. Init initializes some variables including internal state variables, the outputs of integrators and discrete constant blocks, and the auxiliary time variables introduced for managing the execution time of the whole model. Discrete represents the transformed process of discrete blocks of $\mathcal{D}$, and $\dot{\mathbf{y}} = \Gamma(\mathbf{x})$ is the combined vector of the ODEs for all integrator blocks after variable substitution corresponding to other non-integrator continuous blocks; TimeUpdate defines the update of the auxiliary time variables after each loop. The loop period *period*, constraining the domain of the ODEs, is the great common divisor of sample times of all discrete blocks of $\mathcal{D}$.

```
1  module P():
2  output HeatCost = Costin * cost,
       Temperatures = [9 / 5 *
       HouseTemp + 32, 50 + 15 *
       sin(0.262 * t)];
3  begin
4    t := 0;
5    _tick := 0;
6    F2Cin := 70;
7    Thermostat_sub_Relay1_state := 0;
8    tt := 0;
9    Costin := 0;
10   HouseTemp := 20;
11   {
12     F2Cout := 5 / 9 * (70 - 32);
13     Terr := F2Cout - HouseTemp;
14     blowercmd := (if Terr > 5*(5/9)
         then 1 else (if Terr <
         -5*(5/9) then 0 else
         Thermostat_sub_Relay1_state))
15     Thermostat_sub_Relay1_state :=
         blowercmd;
16     {tt_dot = 1, Costin_dot =
         blowercmd * ((-HouseTemp +
         50) * (Mdot * c)),
         HouseTemp_dot = (blowercmd *
         ((-HouseTemp + 50) * (Mdot *
         c)) - (HouseTemp - 5 / 9 *
         (50 + 15 * sin(0.262 * t) -
         32)) * (1 / Req)) * (1 / (M
         * c)) & tt < 0.001}
17     t := t + tt;
18     _tick := _tick + 1;
19     tt := 0;
20   }*
21 end endmodule
22 system P=P() endsystem
```

Fig. 4. The HCSP Model of ITCS

As presented in Fig. 4, the output list (Line 1) includes *HeatCost* representing the cost of heat, and the joint *Temperatures* for the indoor and outdoor temperatures, that correspond to the outputs of the S/S model. Each of the outputs is assigned to their respective values. The main body (Lines 4–20) implements the whole S/S diagram. It starts from the initialisation of a sequence of variables, among which *Costin* and *HouseTemp* are the outputs of two integrator blocks, *F2Cin* the output of a discrete constant block, *Thermostat_sub_Relay1_state* the internal state of Relay block, and *tt*, *t*, *_tick* the auxiliary time variables, then followed by a repetition process. At each round of the repetition, first the discrete blocks of the case study are executed in a correct order, then the two ODEs defining the derivatives of *Costin* and *HouseTemp*, plus the one with *tt*

recording the execution time of each round, are put together to constitute the transformed process of the continuous part. Notice that variable substitution is performed on the right hand sides of each ODE, by replacing recursively the outputs of each non-integrator continuous block as functions of its inputs, till the equations only contain ODE variables and variables from the separate discrete part, e.g. *blowercmd* and *HouseTemp* occurring in the ODEs. Here 0.001 is the period of the whole diagram, and variables t and $\_tick$ represent the accumulated execution time and the number of execution loops respectively.

Notice that the equation of *HouseTemp* depends on the value of *blowercmd*, which is either 0 or 1, the output of Thermostat subsystem. We will consider the two different cases separately in the verification of ITCS.

4.2 Simulation

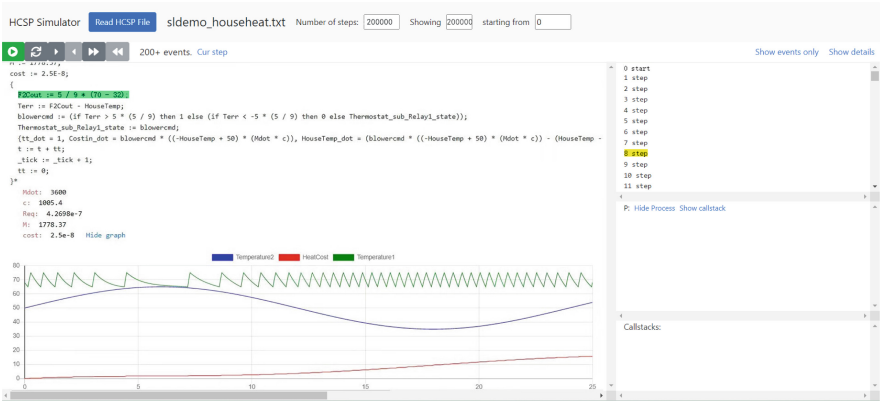


Fig. 5. Simulator interface after importing the HCSP model

MARS guarantees that the generated HCSP model and the corresponding source S/S diagram are consistent by formally defining their semantics and building the equivalence between them [28]. But for visualizing the behavior of the generated HCSP model, we also utilize HCSP simulator integrated in MARS to analyze the generated HCSP model, which reflects the behavior of the source S/S model as expected. HCSP simulator is designed to calculate the execution paths of HCSP processes and visualize them in the graphical interface. As shown in Fig. 5, the left "Read HCSP file" button is used to load the input HCSP models, and on the right, the number of simulation steps, starting position, and ending position can be set. On the left side of the interface, the loaded HCSP process is displayed, with the current executed statement highlighted, and below it, the values of process variables changing over time are displayed; On the right, the trace of all events produced during the execution is shown, including discrete steps, time progress or communication events.

Figure 5 shows the simulation result of the HCSP model of ITCS, by setting the step size to 0.001 s and number of steps to 200,000 respectively. Through the result, we can check whether its behavior aligns with expectation. On the left bottom of Fig. 5, the lines from top to bottom represent the indoor temperature change, the outdoor temperature change, and the cost incurred after the heater is turned on, respectively. The changes and fluctuations of the three curves conform to the design requirement, especially, the house temperature is always within a safe range, to be given in detail in the following verification part.

4.3 Verification

To remedy incomplete simulation, the verification of the HCSP model is needed to guarantee the design requirement strictly. In MARS, this is achieved by HHL Prover through a Hoare-logic style deductive verification method [11,31]. HHL Prover contains three parts: HHLPy [20] for deductive verification of sequential HCSP processes covering ODEs, achieving automation of verification through the annotation of differential and loop invariants and the integration with SMT solvers for solving logical formulas; the interactive Isabelle prover, that is implemented for the whole HCSP with concurrency and communication, conducting proof of HCSP specifications with pre-/post-conditions by manually choosing corresponding inference rules; and the invariant generation, which synthesizes differential invariants for reasoning about ODEs through template-based methods [12,19], to be employed for the former two provers if necessary.

This case study demonstrates the procedure for verifying a safety specification: "if the initial house temperature is within the range of 165/9 °C to 215/9 °C, it will always remain between 145/9 °C and 235/9 °C (i.e. *the safe range*)". For simplicity, the initial system state is defined by the region $165/9 \leq T \leq 215/9$, while the unsafe region is $T \leq 145/9 \vee 235/9 \leq T$, where T stands for *HouseTemp*. Our objective is to verify that all system trajectories originating from the initial region will never enter the unsafe region.

For continuous-time systems, the most challenging part of Hoare-style reasoning is the synthesis of differential invariants. A differential invariant is a set of states Ω satisfying the following three conditions: (1) the initial region is contained in Ω (*Initial Condition*); (2) the unsafe region and Ω are disjoint (*Saturation Condition*); (3) Ω keeps continuous (differential) inductiveness, i.e., all trajectories starting from Ω remain within Ω (*Differential Inductive Condition*). In the following, we adopt a template-based approach for synthesizing a desired differential invariant. Please refer to [12,19,22,25] for more detailed introduction.

Step 1: Simplifying System. From the HCSP model, we see that the system consists of two modes, depending on whether *blowercmd* is 1 or 0, where the dynamics of the temperature are described by

$$\dot{T} = -3.334 \cdot T + 10.916 \cdot \sin(0.262t) + 114.315, \qquad \text{(mode 1)}$$

and

$$\dot{T} = -1.310 \cdot T + 10.916 \cdot \sin(0.262t) + 13.099, \qquad \text{(mode 2)}$$

respectively. The system will switch to mode 1 when $T < 165/9$, and switch to mode 2 when $T > 215/9$.

Note that the above expressions contain a trigonometric function, which will be difficult to reason about. To address this issue, inspired by [13], two fresh variables v and u are introduced to represent $\sin(0.262t)$ and $\cos(0.262t)$. Then, both mode 1 and mode 2 can be transformed to a 3-dimensional system in variables (T, v, u) with polynomial dynamics as follows:

$$\begin{pmatrix} \dot{T} \\ \dot{v} \\ \dot{u} \end{pmatrix} = \begin{pmatrix} -3.334 \cdot T + 10.916 \cdot v + 114.315 \\ 0.262 \cdot u \\ -0.262 \cdot v \end{pmatrix}, \qquad \text{(mode 1')}$$

and

$$\begin{pmatrix} \dot{T} \\ \dot{v} \\ \dot{u} \end{pmatrix} = \begin{pmatrix} -1.310 \cdot T + 10.916 \cdot v + 13.099 \\ 0.262 \cdot u \\ -0.262 \cdot v \end{pmatrix}. \qquad \text{(mode 2')}$$

Note that the derivative $\dot{v}$ is obtained by $\dot{v} = \frac{\mathrm{d}\sin(0.262t)}{\mathrm{d}t} = 0.262\cos(0.262t) = 0.262u$. The computation of $\dot{u}$ is similar. Moreover, we add a new constraint $u^2 + v^2 = 1$ as a state space constraint because $\sin^2(x) + \cos^2(x) = 1$.

Step 2: Setting Templates. Template-based synthesis leverages parameterization. A parameterized template for the target differential invariant is predefined, and constraints are formulated based on the definition of differential invariants. Satisfying these constraints yields the desired differential invariant.

For continuous-time systems, *barrier certificates* [19] are commonly employed for synthesizing differential invariants more efficiently. A barrier certificate is a continuously differentiable function Φ in system variables such that $\Phi \leq 0$ constitutes a differential invariant. For computability, barrier certificates are typically set to be polynomial functions.

Given our system's two polynomial subsystems, mode 1' and mode 2', we introduce two polynomial barrier certificates, Φ_1 and Φ_2, respectively. Each Φ_i is a parameterized polynomial of a user-specified degree d in variable T, v, u of the following form

$$\Phi_i = \sum_{\alpha_1+\alpha_2+\alpha_3 \leq d} c_{i,\alpha_1,\alpha_2,\alpha_3} T^{\alpha_1} v^{\alpha_2} u^{\alpha_3} \quad \text{for } i = 1, 2, \qquad (2)$$

where $\alpha_1, \alpha_2, \alpha_3 \in \mathbb{N}$ are exponents and $c_{i,\alpha_1,\alpha_2,\alpha_3}$ are unknown real coefficients.

Step 3: Solving Constraints. Similar to [9, Theorem 2], the constraints for Φ_1 and Φ_2 are given as follows, for $i = 1, 2$,

$$\forall(T, v, u).\ 165/9 \le T \le 215/9 \wedge u^2 + v^2 = 1 \implies \Phi_i(T, v, u) \le 0, \tag{3}$$

$$\forall(T, v, u).\ (T < 145/9 \vee T > 235/9) \wedge u^2 + v^2 = 1 \implies \Phi_i(T, v, u) > 0, \tag{4}$$

$$\forall(T, v, u).\ u^2 + v^2 = 1 \implies \mathfrak{L}_{f_i}\Phi_i(T, v, u) \le \lambda_i \Phi_i(T, v, u), \tag{5}$$

$$\forall(T, v, u).\ u^2 + v^2 = 1 \implies \Phi_{3-i}(T, v, u) \le \mu_i \Phi_i(T, v, u), \tag{6}$$

where λ_i is any real number, μ_i is any non-negative real number, and $\mathfrak{L}_{f_i}(\Phi_i) = \langle \nabla \Phi_i, f_i \rangle$ is the Lie derivative of function Φ_i w.r.t. f_i (here, ∇ is the gradient notation and $\langle \cdot, \cdot \rangle$ is the dot product). Intuitively, constraint (3) ensures the initial region is contained within the region defined by $\Phi_i(T, v, u) \le 0$, while (4) guarantees exclusion of the unsafe region from this region. Constraint (5) establishes that $\Phi_i(T, v, u) \le 0$ is a differential invariant, ensuring mode i's safety. Finally, (6) maintains safety during mode switches. In our experiments, we set $\lambda_1 = \lambda_2 = -1$, $\mu_1 = \mu_2 = 10$, and the degree of barrier certificate templates to be 2.

To solve these constraints, we employ sum-of-squares optimization techniques to transform them into a hierarchy of semidefinite programming (SDP) relaxations, as detailed in [9,22,25]. In our experiments, we formulate the SDP constraints using Julia package TSSOS [21] and solve them using the Mosek solver [16]. We obtain the following solutions for Φ_1 and Φ_2:

$$\begin{aligned}\Phi_1 =& 0.00553 \cdot T^2 - 0.00536 \cdot T \cdot v + 0.00014 \cdot T \cdot u + 0.69343 \cdot v^2 - 0.00037 \cdot v \cdot u \\ &+ 0.68972 \cdot u^2 - 0.22240 \cdot T + 0.11975 \cdot v - 0.00345 \cdot u + 1.38315, \\ \Phi_2 =& -0.00278 \cdot T^2 - 0.00949 \cdot T \cdot v - 0.15400 \cdot v^2 - 0.00242 \cdot v \cdot u \\ &- 0.15705 \cdot u^2 + 0.07491 \cdot T + 0.15363 \cdot v - 0.00146 \cdot u - 0.31105.\end{aligned}$$

The above synthesized invariants ensure the design requirement.

Currently, the differential invariant generation procedure for HCSP is not fully automated, as it often requires manual template refinement. Once a suitable differential invariant is found, it can be used in HHLPy or Isabelle provers of HCSP for further verification. But for this case study, the synthesized invariants ensure the design requirement directly, thus no further verification is needed.

4.4 Code Generation

The MARS toolchain supports the automatic code generation from HCSP model to C, with correctness guarantee, i.e. the generated C code and the source HCSP model are proved to satisfy the approximate bisimulation relation between their reachable states with given precision allowed in ODE discretization [24]. As a result, the safety properties (that can be considered as sets of system states) proved for the HCSP model are preserved for the generated code with tolerance of given precision. No more verification needs to be re-done at the code level.

For this case study, given any precision $\epsilon > 0$ allowed by the house temperature, our tool can generate the C code that is guaranteed to be approximate bisimilar with the HCSP model thus the original S/S model satisfies the given design requirement that the house temperature is always within the safe range with ϵ tolerance, i.e. (145/9 °C-ϵ, 235/9 °C+ϵ).

By using MARS, the C code for ITCS is generated, part of which is presented in Fig. 6. The whole C implementation of ITCS consists of 97 lines, that is significantly less than the code automatically generated from S/S (to be shown later). Figure 6 presents the discretization code corresponding to the ODE part (Line 16 of Fig. 4), where h is the discretized step with respect to given precision. The ODEs of *tt*, *Costin*, *HouseTemp* are discretized using Runge-Kutta method, implemented by a while loop: a sequence of discrete assignments on calculating the approximate values of continuous variables is performed in each loop, and when the boundary of the ODEs is reached, the loop breaks. The system's running results can be observed by executing the generated C code. Among the results in Fig. 8, we can see that the execution results of the generated C code from HCSP are almost identical to the ones of the HCSP model.

```
1  while (1) {
2      double tt_ori = tt;
3      double Costin_ori = Costin;
4      double HouseTemp_ori = HouseTemp;
5      double tt_dot1 = 1;
6      double Costin_dot1 = blowercmd *
           ((-HouseTemp + 50) * (Mdot *
           c));
7      double HouseTemp_dot1 =
           (blowercmd * ((-HouseTemp +
           50) * (Mdot * c)) -
           (HouseTemp - 5 / 9 * (50 +
           15 * sin(0.262 * t) - 32)) *
           (1 / Req)) * (1 / (M * c));
8      tt = tt_ori + tt_dot1 * h / 2;
9      Costin = Costin_ori +
           Costin_dot1 * h / 2;
10     HouseTemp = HouseTemp_ori +
           HouseTemp_dot1 * h / 2;
11     ... ...
12     tt = tt_ori + (tt_dot1 + 2 *
           tt_dot2 + 2 * tt_dot3 +
           tt_dot4) * h / 6;
13     Costin = Costin_ori +
           (Costin_dot1 + 2 *
           Costin_dot2 + 2 *
           Costin_dot3 + Costin_dot4) *
           h / 6;
14     HouseTemp = HouseTemp_ori +
           (HouseTemp_dot1 + 2 *
           HouseTemp_dot2 + 2 *
           HouseTemp_dot3 +
           HouseTemp_dot4) * h / 6;
15     delay(threadNumber, h);
16     if (!(tt < 0.001)) {
17         break;
18     }
19 }
```

Fig. 6. Part of the C code generated from MARS

```
1  void ODEUpdateStates(SolverInfo *si)
2  {/* the solver specified*/}
3  void house_step()
4  { ...
5    if (IsMajorTimeStep(house)) {
6      ODEUpdateStates(&house->solver);
7    ... ...
8  }
9  void main()
10 { ...
11     house_initialize();
12     while ((rtmGetErrorStatus(house)
           == (NULL)) &&
13           !rtmGetStopRequested(house))
               {
14     house_step();
15   }
16   house_terminate();
17   ...
18 }
```

Fig. 7. Part of the C code generated from Simulink

We also use S/S to generate the C code of ITCS, which amounts to 382 lines in total and is partly shown in Fig. 7. It includes the functions for updating continuous states with the specified solver for ODEs, the step function for executing the whole model, and the main function that initializes, steps in a while loop, and terminates the execution in sequence. The reason for the lengthy code from S/S includes: on one hand, the HCSP model transformed from the S/S diagram combines all the blocks of each connected part of the diagram with integrator blocks into one ODE vector, through variable substitution by hiding all outputs of intermediate non-integrator blocks, and as a result, the C code generated from the HCSP model will not include the local assignments corresponding to these blocks; On the other hand, S/S Coder needs to do settings related to ODE solver types, data logging, *etc.* for each S/S instance, while in our tool, all these settings are determined, which indeed lacks feasibility for some extreme cases, but promotes efficiency for normal cases that can be handled using the general ODE solver based on Runge-Kutta method. We will give a more comparison in next section. Figure 8 presents the comparison of execution results of the C code generated from HCSP model and S/S, the HCSP model and the original S/S model respectively. We can see that all of them are mostly consistent, except for some small fluctuations.

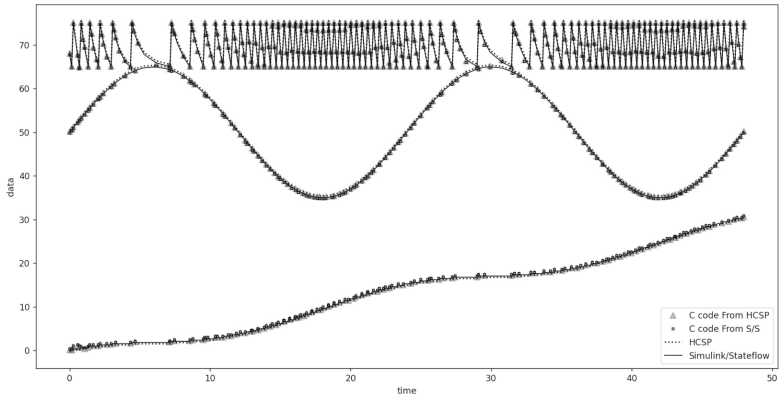


Fig. 8. Comparison of the execution results

5 Comparison with S/S

We compare our approach with S/S from three aspects throughout model-based development of systems: modeling and analysis, verification, code generation.

Modeling and Analysis. Based on a rich set of individually simple blocks and their hierarchical composition, S/S offers a powerful graphical modeling language

for building embedded systems. Especially, it is capable of modeling dynamic systems involved with continuous physical plants and complex control logics. S/S does not have an official formal semantics, and instead, system analysis and design validation within S/S are based on numerical simulation, which provides a variety of ODE solvers especially the varying-step ones for solving ODEs with both efficiency and accuracy. MARS reuses S/S for the graphical modeling of software functionality and continuous plants of systems, and to remedy S/S, it further integrates AADL for the modeling of architectures [27]. MARS also provides HCSP language for the formal modeling of hybrid systems, with formal semantics defined, and supports the transformation from S/S diagrams to HCSP formal models. The transformation covers a subset of S/S graphical syntax related to the design of hybrid systems, which is also the focus of HCSP, and the correctness of the transformation is guaranteed by defining both the formal semantics of S/S and HCSP and proving their bisimulation between each other. MARS implements a HCSP simulator that invokes Python's Scipy package to have fixed-step solvers for solving ODEs.

Verification. Formal verification is necessary in the development process of safety-critical systems. S/S has a well integrated commercial verification toolset called Simulink Design Verifier (SLDV) [14], which offers static analysis and discrete-time verification of S/S models with a high degree of automation. However, same as Simulink, the verifier does not have a formal specification language with formal semantics, and instead, it represents the property to be proved also as a S/S model. As a result, the result of SLDV cannot guarantee soundness. In [17], the authors use SLDV to formally verify an automotive Simulink controller model and detect some bugs of SLDV. MARS reduces the verification of S/S models to the verification of the transformed HCSP formal model, due to the consistency guarantee of the transformation from S/S to HCSP. As shown in Fig. 1, MARS integrates HCSP verification tools based on a sound hybrid Hoare logic for reasoning about HCSP, implemented via interactive and automated theorem proving. Furthermore, it is able to reason about continuous time behavior of HCSP models involved with ODEs based on differential invariant generation. Due to the complexity of hybrid systems, the verification of HCSP related to invariant synthesis, communications and parallel composition, needs to be done manually, but its soundness and capability of handling these behaviors are very important for designing safety-critical control systems.

Code Generatation. In the previous section, we have already made some comparison between S/S and our approach for code generation of the case study. S/S has an integrated code generator, which is well developed and applied to many scalable practical embedded systems. However, the auto-generated C code may differ from the behavior of the original S/S model due to the lack of formal semantics, or potential bugs in the translation procedure from S/S to C. Thus, the code generated from S/S needs further verification for the safety. In [3], the authors perform formal verification of C code that is automatically generated from S/S controller models and find errors that are not inconsistent with the design requirement. Although these errors are found to exist as well for the orig-

inal S/S model, it does not mean that the translation is correct, and in contrary, it shows the consequence of the original S/S lacking formal semantics and verification. In our tool, we implement a formally verified code generator from HCSP to C, which solves the above problem, but honestly, it is challenging for our tool to be applied for the development of large scale systems (mostly due to the verification intrinsic difficulty of complex systems).

6 Conclusion

In this paper, we show how to design an intelligent temperature control system with MARS. It consists of a S/S graphical model, a HCSP formal model transformed from the graphical model, the simulation and verification of the HCSP model, and the C code automatically generated from the verified HCSP formal model. Compared with the development of the system with S/S, the advantages of the design of CPS with MARS include: i) the correctness and reliability of the generated C code; ii) integration of modeling, simulation, verification and code generation, as well as integration of formal and informal design for CPS.

Acknowledgements. This work has been partially funded by the National Key R&D Program of China under grant No. 2022YFA1005100, 2022YFA1005101, and 2022YFA1005104, by the NSFC under grant No. 62192732 and 62032024, the CAS Project for Young Scientists in Basic Research under grant No. YSBR-040, and the Major Project of ISCAS (ISCAS-ZD-202302).

References

1. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.-H.: Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In: Grossman, R.L., Nerode, A., Ravn, A.P., Rischel, H. (eds.) HS 1991-1992. LNCS, vol. 736, pp. 209–229. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-57318-6_30
2. Ansys Inc.: Esterel Technologies, SCADE Suite (2018). http://www.esterel-technologies.com/products/scade
3. Berger, P., Katoen, J.-P., Ábrahám, E., Waez, M.T.B., Rambow, T.: Verifying auto-generated C code from Simulink. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E. (eds.) FM 2018. LNCS, vol. 10951, pp. 312–328. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-95582-7_18
4. Chen, X., Sankaranarayanan, S., Ábrahám, E.: Under-approximate flowpipes for non-linear continuous systems. In: FMCAD 2014, pp. 59–66 (2014)
5. Feiler, P., Gluch, D.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley (2012)
6. Frehse, G., et al.: SpaceEx: scalable verification of hybrid systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_30
7. He, J.: From CSP to hybrid systems. In: A Classical Mind, pp. 171–189. Prentice Hall International (UK) Ltd. (1994)

8. Henzinger, T.A.: The theory of hybrid automata. In: LICS 1996, pp. 278–292. IEEE Computer Society (1996)
9. Kong, H., He, F., Song, X., Hung, W.N.N., Gu, M.: Exponential-condition-based barrier certificate generation for safety verification of hybrid systems. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 242–257. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_17
10. Kong, S., Gao, S., Chen, W., Clarke, E.: dReach: σ-reachability analysis for hybrid systems. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 200–205. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_15
11. Liu, J., et al.: A calculus for hybrid CSP. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 1–15. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17164-2_1
12. Liu, J., Zhan, N., Zhao, H.: Computing semi-algebraic invariants for polynomial dynamical systems. In: EMSOFT 2011, pp. 97–106 (2011)
13. Liu, J., Zhan, N., Zhao, H., Zou, L.: Abstraction of elementary hybrid systems by variable transformation. In: Bjørner, N., de Boer, F. (eds.) FM 2015. LNCS, vol. 9109, pp. 360–377. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19249-9_23
14. MathWorks Inc.: Simulink Design Verifier – User's guide. https://de.mathworks.com/help/pdf_doc/sldv/sldv_ug.pdf
15. MathWorks Inc.: Simulink User's Guide (2013). http://www.mathworks.com/help/pdf_doc/simulink/sl_using.pdf
16. MOSEK ApS: MOSEK Optimizer API for Julia. Version 10.1.13 (2019). https://docs.mosek.com/latest/juliaapi/index.html
17. Nellen, J., Rambow, T., Waez, M.T.B., Ábrahám, E., Katoen, J.-P.: Formal verification of automotive Simulink controller models: empirical technical challenges, evaluation and recommendations. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E. (eds.) FM 2018. LNCS, vol. 10951, pp. 382–398. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-95582-7_23
18. Platzer, A.: Differential dynamic logic for hybrid systems. J. Autom. Reason. **41**(2), 143–189 (2008)
19. Prajna, S., Jadbabaie, A.: Safety verification of hybrid systems using barrier certificates. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 477–492. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24743-2_32
20. Sheng, H., Bentkamp, A., Zhan, B.: HHLPy: practical verification of hybrid systems using Hoare logic. In: Chechik, M., Katoen, J.P., Leucker, M. (eds.) FM 2023. LNCS, vol. 14000, pp. 160–178. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-27481-7_11
21. Wang, J., Magron, V., Lasserre, J.: TSSOS: a moment-SOS hierarchy that exploits term sparsity. SIAM J. Optim. **31**(1), 30–58 (2021)
22. Wang, Q., Chen, M., Xue, B., Zhan, N., Katoen, J.: Encoding inductive invariants as barrier certificates: synthesis via difference-of-convex programming. Inf. Comput. **289**(Part), 104965 (2022)
23. Wang, S., Zhan, N., Zou, L.: An improved HHL prover: an interactive theorem prover for hybrid systems. In: Butler, M., Conchon, S., Zaïdi, F. (eds.) ICFEM 2015. LNCS, vol. 9407, pp. 382–399. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25423-4_25
24. Wang, S., Ji, Z., Xu, X., Zhan, B., Gao, Q., Zhan, N.: Formally verified C code generation from hybrid communicating sequential processes. In: ICCPS 2024, pp. 123–134. IEEE (2024)

25. Wu, H., Feng, S., Gan, T., Wang, J., Xia, B., Zhan, N.: On completeness of SDP-based barrier certificate synthesis over unbounded domains. In: Platzer, A., Rozier, K.Y., Pradella, M., Rossi, M. (eds.) FM 2024. LNCS, vol. 14934, pp. 248–266. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-71177-0_16
26. Xu, X., Talpin, J., Wang, S., Zhan, B., Zhan, N.: Semantics foundation for cyber-physical systems using higher-order UTP. ACM Trans. Softw. Eng. Methodol. **32**(1), 9:1–9:48 (2023)
27. Xu, X., Wang, S., Zhan, B., Jin, X., Talpin, J., Zhan, N.: Unified graphical co-modeling, analysis and verification of cyber-physical systems by combining AADL and Simulink/Stateflow. Theor. Comput. Sci. **903**, 1–25 (2022)
28. Xu, X., Zhan, B., Wang, S., Talpin, J.P., Zhan, N.: A denotational semantics of Simulink with higher-order UTP. J. Log. Algebraic Methods Program. **130**, 100809 (2023)
29. Yan, G., Jiao, L., Wang, S., Wang, L., Zhan, N.: Automatically generating SystemC code from HCSP formal models. ACM TOSEM **29**(1), 4:1–4:39 (2020)
30. Zhan, B., et al.: Mars 2.0: a toolchain for modeling, analysis, verification and code generation of cyber-physical systems. arXiv abs/2403.03035 (2024)
31. Zhan, N., Zhan, B., Wang, S., Guelev, D.P., Jin, X.: A generalized hybrid Hoare logic. CoRR abs/2303.15020 (2023)