A Proof System for Cyber-physical Systems with Shared-Variable Concurrency

Ran Li[1], Huibiao Zhu[1(✉)], and Richard Banach[2]

[1] Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China
hbzhu@sei.ecnu.edu.cn
[2] Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, UK

Abstract. Cyber-physical system (CPS) is about the interplay of discrete behaviors and continuous behaviors. The combination of the physical and the cyber may cause hardship for the modeling and verification of CPS. Hence, a language based on shared variables was proposed to realize the interaction in CPS. In this paper, we formulate a proof system for this language. To handle the parallel composition with shared variables, we extend classical Hoare triples and bring the trace model into our proof system. The introduction of the trace may complicate our specification slightly, but it can realize a compositional proof when the program is executing. Meanwhile, this introduction can set up a bridge between our proof system and denotational semantics. Throughout this paper, we also present some examples to illustrate the usage of our proof system intuitively.

Keywords: Cyber-physical System (CPS) · Shared Variables · Trace Model · Hoare Logic.

1 Introduction

Cyber-physical system (CPS) [6,7] is an integration of discrete computer control behaviors and continuous physical behaviors. In CPS, computer programs can influence physical behaviors, and vice versa. It has covered a wide range of application areas, including healthcare equipment, intelligent traffic control and environmental monitoring, etc.

The interaction between the cyber and the physical brings convenience for many applications, while it may also complicate the design and the modeling of systems. Thus, some specification languages are proposed to describe and model CPS. Henzinger described hybrid systems with hybrid automata [3]. Zhou et al. developed a language called Hybrid CSP [12] which supports parallel composition via the communication mechanism, and Liu et al. proposed a calculus for it in [9]. He et al. presented a hybrid relational modeling language (HRML) in [2]. Different from them, we proposed a language whose parallel mechanism is based on shared variables in our previous work [1]. Further, we elaborated this language and proposed its denotational and algebraic semantics in [8].

In this paper, we give a proof system for this language based on our previous work [1, 8]. The major challenge of formulating our proof system is how to cope with the parallel programs that contain shared variables. For parallel programs supported by shared variables, there are two main classical verification methods. In the well-known Owicki&Gries system [11], they defined interference-free to realize parallel composition. However, it may require proving numerous assertions when we check the property of interference-free. Therefore, a compositional method called rely-guarantee was proposed in [5]. This approach adds a rely condition and a guarantee condition in its specification, so that compositionality is achieved. In [10], Lunel et. al. employed this approach to present a component-based verification technique for Computer Controlled System (CCS) in differential dynamic logic. Nevertheless, it takes some effort to determine the corresponding rely condition and guarantee condition.

Slightly different from the two ways, we introduce the trace model into our proof system. For the feature of shared variables in the parallel mechanism, we define a *Merge* function to solve it. Thanks to the trace model, programs can perceive environment's actions and they can be composed during their executions consequently. However, the trace model is not a panacea. We add the corresponding preconditions and postconditions (similar to those in Hoare Logic [4]) to record the state of continuous variables, in that the trace only tracks values of discrete variables. Further, we attach the global clock variable *now* to our proof system. Therefore, it can capture the real-time feature of CPS. Altogether, we extend the traditional triple in Hoare Logic [4] $\{p\}\ S\ \{q\}$ to $[tr \bullet p_c]\ S\ [tr' \bullet q_c]$. tr and tr' are responsible for recording values of discrete variables. p_c and q_c contain the pre/postcondition of continuous variables and the starting/termination time of S. Because the focus of the proof is the initial and terminal data state of the program, we present a transformation rule to build the bridge between our proof system and traditional Hoare Logic.

Although the introduction of the trace model complicates the expression form slightly, it realizes compositional proof during the programs' executions which cannot be done in Owicki&Gries system. What's more, since the trace model records all data states, it can provide more precise conditions of the environment compared with the rely-guarantee approach. Moreover, our approach can link our proof system with denotational semantics, since the trace model is used in the description of our denotational semantics [8]. Overall, we give several rules for basic commands and compound constructs in this paper. Besides, to aid the understanding of our proof system, we also apply it to an example of a battery management system.

The remainder of this paper is organized as follows. In Section 2, we recall the syntax of this language and introduce the trace model briefly. Based on the feature of this language and trace model, the specification of our proof system is given. In Section 3, we list a set of rules used to specify and prove the correctness of CPS. To showcase the application of our proof system, Section 4 is dedicated to the example of a battery management system. We conclude our work and discuss some future work in Section 5.

2 Semantic Model

In this section, we first recall the syntax of the language to describe CPS. Then, we introduce the trace model to support our proof system in the next section. On this basis, the specification of our proof system is given as well.

2.1 Syntax of CPS with Shared-Variable Concurrency

As shown in Table 1, we follow the syntax proposed and elaborated in [1,8]. Here, x is a discrete variable and v is a continuous variable. b stands for a Boolean expression and e represents a discrete or continuous expression. The process in our language contains discrete behaviors Db, continuous behaviors Cb and various compositions of Db and Cb.

Table 1. Syntax of CPS

Process	$P, Q ::= Db$	(Discrete behavior)
	$\mid Cb$	(Continuous behavior)
	$\mid P; Q$	(Sequential Composition)
	$\mid$ **if** b **then** P **else** Q	(Conditional Construct)
	$\mid$ **while** b **do** P	(Iteration Construct)
	$\mid P \parallel Q$	(Parallel Composition)
Discrete behavior	$Db ::= x := e \mid @gd$	
Continuous behavior	$Cb ::= R(v, \dot{v})$ **until** g	
Guard Condition	$g ::= gd \mid gc \mid gd \vee gc \mid gd \wedge gc$	
Discrete Guard	$gd ::= true \mid x = e \mid x < e \mid x > e \mid gd \vee gd \mid gd \wedge gd \mid \neg gd$	
Continuous Guard	$gc ::= true \mid v = e \mid v < e \mid v > e \mid gc \vee gc \mid gc \wedge gc \mid \neg gc$	

- **Db:** There are two discrete actions in our language.
  - $x := e$ is a discrete assignment and it is an atomic action. Through this assignment, the expression e is evaluated and the value gained is assigned to the discrete variable x.
  - $@gd$ is a discrete event guard. It is triggered if gd is satisfied. Otherwise, the process waits for the environment to trigger gd. Note that the environment stands for the other program given by the parallel composition. For example, if the whole system is $P \parallel Q$ and we now analyze the program P, then Q is P's environment.
- **Cb:** To define the continuous behaviors in CPS, we introduce differential relation in our language.
  - $R(v, \dot{v})$ **until** g is the syntax of describing continuous behavior. $R(v, \dot{v})$ is a differential relation which defines the dynamics of the continuous variable v. The evolution of v will follow the differential relation until the guard condition g is triggered.
- **Composition:** Also, a process can be comprised of the above commands.
  - $P; Q$ is sequential composition. The process Q is executed after the process P's successful termination.
  - **if** b **then** P **else** Q is a conditional construct. If the Boolean condition b is true, then P is executed. Otherwise, Q is executed.

- **while** b **do** P is an iteration construct. P keeps running repeatedly until the Boolean condition b does not hold.
- $P \parallel Q$ is parallel composition. It indicates P executes in parallel with Q. The parallel mechanism is based on shared variables. In our language, shared writable variables only focus on discrete variables.

2.2 Trace Model

The parallel mechanism in our language is based on shared variables. We introduce a trace model to record the communication during their execution. Trace is defined to describe the behavior of a program, and it is composed of a series of snapshots. A snapshot specifies the behavior of an atomic action, and it is expressed as a triple (t, σ, μ).

- t: It records the time when the action occurs.
- σ: We use σ to record the states of data (i.e., discrete variables) contributed by the program itself or its environment (i.e., the other program given by the parallel composition) during the program's runtime.
- μ: We introduce it to indicate whether the action is done by the program itself or by the environment. If $\mu = 1$, it means that this action is performed by the program itself. If $\mu = 0$, it implies that this action is contributed by the environment. The introduction of μ can support the data exchange for the components of a parallel process with the help of the *Merge* function (shown on page 12).

We list the following projection functions $\pi_i (i = 1, 2, 3)$ to get the ith element of a snapshot.

$$\pi_1((t, \sigma, \mu)) =_{df} t, \quad \pi_2((t, \sigma, \mu)) =_{df} \sigma, \quad \pi_3((t, \sigma, \mu)) =_{df} \mu$$

Further, we also define some operators of traces and snapshots.

- $last(tr)$ stands for the last snapshot of the trace tr.
- $tr_a \frown tr_b$ denotes the concatenation of the trace tr_a and tr_b.
- Assume that tr contains at least one snapshot, the notation $tr - last(tr)$ indicates the rest of tr after deleting the last snapshot of tr.
- Assume that the snapshot sp is not the first one in its trace tr, $pre(sp, tr)$ denotes the previous snapshot of sp in tr.
- $tr = tr_a \vee tr_b$ is equivalent to $tr = tr_a \vee tr = tr_b$. It implies that there are two possible cases of tr. We introduce this notation to represent different execution orders under parallel composition.

2.3 Specification

Based on the trace model, we introduce the formalism to specify and verify CPS with shared-variable concurrency that is modeled by our language.

The classic Hoare triples [4] have the form of $\{p\}\ S\ \{q\}$. p, q and S denote precondition, postcondition and program respectively. It indicates that if the

program S is executed under the precondition p, the final state of S satisfies the postcondition q when S terminates.

For parallelism with shared variables, slightly different from the two existing methods (i.e., interference-free checking [11] and rely-guarantee [5]), we can pave the way for parallel composition in the light of the above trace model. The trace model gives the process an insight into the behaviors of the environment. Whereas, since trace can only record the state of discrete variables in our model, we also need to utilize traditional precondition and postcondition to mark values of continuous variables. Besides, time is a crucial element in CPS, so we add a global clock variable called *now* in our assertions.

Altogether, the formula has the new form of $[tr \bullet p_c]\ S\ [tr' \bullet q_c]$. tr and tr' represent the initial trace and ending trace. p_c and q_c record the states of continuous variables and the starting/termination time of S.

3 Proof System

In this section, we present a proof system for our language to prove the correctness of CPS with shared-variable concurrency. Also, some programs are attached and act as examples to illustrate the usage of these rules.

3.1 Auxiliary Rules

We give two auxiliary rules in this subsection. **Rule 1** is defined to convert the formula of the trace form to the classic form in Hoare Logic. **Rule 2** is similar to the traditional consequence rule in Hoare Logic.

Rule 1. Transformation

$$\frac{[tr \bullet p_c]\ S\ [tr' \bullet q_c],\ (last(tr) \bullet p_c) \rightsquigarrow p,\ (last(tr') \bullet q_c) \rightsquigarrow q}{\{p\}\ S\ \{q\}}$$

Actually, when we prove the correctness of program, the real focus is data state. Thus, we link our proof system with traditional Hoare Logic through this rule. We introduce the notation of $\rightsquigarrow$ as below. It maps the elements of the snapshot sp to the corresponding data states. $PL(\sigma)$ translates the mapping relations in the data state into expressions of predicate logic. For example, if $\sigma =_{df} \{x \mapsto 0, y \mapsto 1\}$, then $PL(\sigma) = (x = 0 \wedge y = 1)$.

$$(sp \bullet p_c) \rightsquigarrow p =_{df} (PL(\pi_2(sp)) \wedge p_c) \rightarrow p$$

Rule 2. Consequence

$$\frac{[tr_1 \bullet p_{c1}]\ S\ [tr_1' \bullet q_{c1}],\ (tr \bullet p_c) \xrightarrow{c} (tr_1 \bullet p_{c1}),\ (tr_1' \bullet q_{c1}) \xrightarrow{c} (tr' \bullet q_c)}{[tr \bullet p_c]\ S\ [tr' \bullet q_c]}$$

We use this rule to strengthen the precondition and weaken the postcondition. The notation of $\xrightarrow{c}$ is used to define the implication relation of data states.

$$\begin{aligned}(tr \bullet p_c) \xrightarrow{c} (tr_1 \bullet p_{c1}) \ =_{df} & ((tr - last(tr)) = (tr_1 - last(tr_1))) \\ & \wedge ((PL(\pi_2(last(tr))) \wedge p_c) \rightarrow (PL(\pi_2(last(tr_1))) \wedge p_{c1}))\end{aligned}$$

3.2 Proof Rules for Basic Commands

3.2.1 Assignment

Rule 3. Assignment

$$\frac{tr \xrightarrow{env} (tr' - last(tr')),\ pre(last(tr'), tr') \rightsquigarrow PL(\pi_2(last(tr')))[e/x]}{[tr \bullet p_c]\ x := e\ [tr' \bullet p_c]}$$

Here, $PL(\sigma)[e/x]$ is the same as $PL(\sigma)$ except that all free instances of variable x are substituted for expression e. We define $\xrightarrow{env}$ to describe the relationship between tr and tr'. It implies that only environment steps are performed between the end of tr and the end of tr'. This is due to the fact that when "x" is a shared variable, it can be modified by the environment's action.

$$tr_1 \xrightarrow{env} tr_2 =_{df} \exists s \cdot (tr_2 = tr_1 \frown s \wedge \forall ttr \in s \cdot \pi_3(ttr) = 0)$$

An example of an application of this rule is:

$$[tr_0 \bullet v = 1.25 \wedge now = 0]\ x := x + 1\ [tr_1 \bullet v = 1.25 \wedge now = 0].$$

Here, $tr_0 =_{df} \langle (0, \sigma_0, 1) \rangle$, $tr_1 =_{df} \langle (0, \sigma_0, 1), (0, \sigma_1, 1) \rangle$, $\sigma_0 =_{df} \{x \mapsto 0\}$ and $\sigma_1 =_{df} \{x \mapsto 1\}$. $v = 1.25 \wedge now = 0$ represents the initial state of continuous variables and the starting time, and it remains unchanged. This is because the discrete assignment does not change continuous values, and it is an atomic action which takes no time.

3.2.2 Discrete Event Guard

For the discrete event guard $@gd$, it can be triggered by the program itself or by the environment, so we divide it into two rules as follows.

(1) Rule 4-1 describes that gd is triggered due to the program's own action.
Rule 4-1. Guard_SelfTrig

$$\frac{last(tr) \rightsquigarrow gd}{[tr \bullet p_c]\ @gd\ [tr \bullet p_c]}$$

$last(tr) \rightsquigarrow gd$ means that the current trace can trigger gd, that is, the event guard gd is triggered by the program itself without waiting for the environment to trigger. Due to this immediate trigger action, nothing needs to change.

By utilizing this rule, we can prove the following formula. Here, $tr_1 =_{df} \langle (0, \sigma_0, 1), (0, \sigma_1, 1) \rangle$, $\sigma_0 =_{df} \{x \mapsto 0\}$ and $\sigma_1 =_{df} \{x \mapsto 1\}$.

$$[tr_1 \bullet v = 1.25 \wedge now = 0]\ @(x > 0)\ [tr_1 \bullet v = 1.25 \wedge now = 0].$$

(2) Rule 4-2 implies gd waits for the environment to trigger it.
Rule 4-2. Guard_EnvTrig

$$\frac{\begin{array}{l} last(tr) \rightsquigarrow \neg gd,\ last(s) \rightsquigarrow gd, \\ \forall ttr \in s \cdot \pi_3(ttr) = 0,\ \forall ttr \in (s - last(s)) \cdot ttr \rightsquigarrow \neg gd \end{array}}{[tr \bullet p_c]\ @gd\ [tr \frown s \bullet p_c[\pi_1(last(s))/now]]}$$

If the process cannot trigger the guard under the current state (i.e., $last(tr) \rightsquigarrow \neg gd$), the process will wait for environment's action to trigger gd. $\forall ttr \in s \cdot$

$\pi_3(ttr) = 0$ implies that all the actions in the trace s are contributed by the environment. $\forall ttr \in (s - last(s)) \cdot ttr \rightsquigarrow \neg gd$ and $last(s) \rightsquigarrow gd$ emphasize that it is the last action of s triggers gd and none of the previous actions can trigger gd. The global clock now is updated to $\pi_1(last(s))$ because of the consumption of waiting.

With this rule, we can deduce:

$$[tr_1 \bullet v = 1.25 \wedge now = 0] \ @(y > 0) \ [tr_2 \bullet v = 1.25 \wedge now = 2].$$

Here, $tr_1 =_{df} \langle (0, \sigma_0, 1), (0, \sigma_1, 1) \rangle$, $tr_2 =_{df} tr_1 ^\frown \langle (1, \sigma', 0), (2, \sigma'', 0) \rangle$, $\sigma_0 =_{df} \{x \mapsto 0\}$, $\sigma_1 =_{df} \{x \mapsto 1\}$, $\sigma' =_{df} \{x \mapsto 1, y \mapsto 0\}$ and $\sigma'' =_{df} \{x \mapsto 1, y \mapsto 1\}$.

3.2.3 Continuous Behavior

For the continuous behavior $R(v, \dot{v})$ **until** g, we list the following four types of rules according to the types of guard g. More specifically, we further detail each type of rules on the basis of the time when g is triggered (i.e., triggered at the beginning or not).

(1) Rule 5-1-1 and **Rule 5-1-2** describe the situation where the guard condition is merely determined by continuous variables (i.e., $g \equiv gc$). Hence, we can omit trace in this rule and focus on the states of continuous variables in them.

Rule 5-1-1. Cb_ContGuard_1

$$\frac{gc(v_{now})}{[tr \bullet p_c] \ R(v, \dot{v}) \ \textbf{until} \ gc \ [tr \bullet p_c]}$$

Rule 5-1-2. Cb_ContGuard_2

$$\frac{\begin{array}{c} p_c[t_0/now] \wedge \exists t' \in (t_0, \infty) \cdot \ (gc(v_{t'}) \wedge \forall t \in [t_0, t') \cdot \neg gc(v_t) \wedge now = t') \\ \wedge R(v, \dot{v}) \ \textbf{during} \ [t_0, now) \rightarrow q_c \end{array}}{[tr \bullet p_c] \ R(v, \dot{v}) \ \textbf{until} \ gc \ [tr \bullet q_c]}$$

As shown in **Rule 5-1-1**, if gc is satisfied at the beginning of the program (i.e., $gc(v_{now})$), then the state will not change. Here, $gc(v_t)$ means that the value of continuous variables v at the time t makes gc true.

Rule 5-1-2 describes the situation where gc is not triggered at the beginning. Here, t_0 is a fresh variable representing the initial time. The program waits for gc to be triggered (highlighted in **Rule 5-1-2**). now is updated to t' and the continuous variable v is evolving as $R(v, \dot{v})$ during this period, expressed by $R(v, \dot{v})$ **during** $[t_0, now)$. By applying this rule, we can get the following proof. Here, $tr_0 =_{df} \langle (0, \sigma_0, 1) \rangle$ and $\sigma_0 =_{df} \{x \mapsto 0\}$.

$$[tr_0 \bullet v = 1.25 \wedge now = 0] \ \dot{v} = 1 \ \textbf{until} \ v \geqslant 2.5 \ [tr_0 \bullet v = 2.5 \wedge now = 1.25].$$

(2) Rule 5-2-1 and **Rule 5-2-2** contain rules that the guard concerns purely discrete variables (i.e., $g \equiv gd$). Different from the above rules, we need to track the traces. This is because the states of discrete variables are recorded in traces.

Rule 5-2-1. Cb_DiscGuard_1

$$\frac{last(tr) \rightsquigarrow gd}{[tr \bullet p_c] \ R(v, \dot{v}) \ \textbf{until} \ gd \ [tr \bullet p_c]}$$

Rule 5-2-2. Cb_DiscGuard_2

$$\frac{p_c[t_0/now] \wedge EnvTrig2(gd) \wedge now = \pi_1(last(s)) \wedge R(v,\dot{v}) \textbf{ during } [t_0, now) \rightarrow q_c}{[tr \bullet p_c]\ R(v,\dot{v}) \textbf{ until } gd\ [tr^\frown s \bullet q_c]}$$

The process is evolving as $R(v,\dot{v})$ until the guard gd is triggered. Similarly, as shown in **Rule 5-2-1**, if gd is triggered when the program starts, then nothing needs to change.

Otherwise, as presented in **Rule 5-2-2**, gd waits to be triggered by the environment, defined by $EnvTrig2(gd)$. The definition of $EnvTrig2(gd)$ below is similar to **Rule 4-2**.

$$EnvTrig2(gd) =_{df} last(tr) \rightsquigarrow \neg gd \wedge last(s) \rightsquigarrow gd \wedge \forall ttr \in s \cdot \pi_3(ttr) = 0 \\ \wedge \forall ttr \in (s - last(s)) \cdot ttr \rightsquigarrow \neg gd$$

This rule leads to:

$$[tr_1 \bullet v = 1.25 \wedge now = 0]\ \dot{v} = 1 \textbf{ until } y > 0\ [tr_2 \bullet v = 3.25 \wedge now = 2].$$

Here, $tr_1 =_{df} \langle (0,\sigma_0,1),(0,\sigma_1,1)\rangle$, $tr_2 =_{df} tr_1{}^\frown\langle (1,\sigma',0),(2,\sigma'',0)\rangle$, $\sigma_0 =_{df} \{x \mapsto 0\}$, $\sigma_1 =_{df} \{x \mapsto 1\}$, $\sigma' =_{df} \{x \mapsto 1, y \mapsto 0\}$ and $\sigma'' =_{df} \{x \mapsto 1, y \mapsto 1\}$.

(3) Rule 5-3-1 and **Rule 5-3-2** denote the condition where the guard is a hybrid one with the form of $gd \wedge gc$.

Rule 5-3-1. Cb_HybridGuard1_1

$$\frac{gc(v_{now}) \wedge (last(tr) \rightsquigarrow gd)}{[tr \bullet p_c]\ R(v,\dot{v}) \textbf{ until } gd \wedge gc\ [tr \bullet p_c]}$$

Rule 5-3-2. Cb_HybridGuard1_2

$$\frac{\begin{array}{l} p_c[t_0/now] \wedge (last(tr^\frown s) \rightsquigarrow gd) \wedge Await(gd \wedge gc) \\ \quad \wedge \exists t' \geqslant \pi_1(last(tr^\frown s)) \cdot (gc(v_{t'}) \wedge now = t') \wedge R(v,\dot{v}) \textbf{ during } [t_0, now) \rightarrow q_c \end{array}}{[tr \bullet p_c]\ R(v,\dot{v}) \textbf{ until } gd \wedge gc\ [tr^\frown s \bullet q_c]}$$

For the hybrid guard $gd \wedge gc$, only when gd and gc are both triggered, the relevant continuous evolution terminates. **Rule 5-3-1** depicts that $gd \wedge gc$ is satisfied at the beginning.

We apply **Rule 5-3-2** to present the condition where $gd \wedge gc$ is not triggered at the beginning. We can first wait for gd to become true. Once gd holds, we wait for gc. $Await(gd \wedge gc)$ defined below states that both guards do not hold in the intervening period, and the highlighted part formalizes gd is not satisfied at t. Since gd can only be changed at discrete time points, it keeps unchanged until the next discrete action happens. As illustrated in Fig. 1, we define $LClosest(tr^\frown s, t)$ (i.e., the closest discrete action's time point to t) to imply gd's state at t. Here, π_1^* is lifted from a single snapshot to a sequence of snapshots. The demonstration of this rule is presented in Fig. 2(a).

$$Await(gd \wedge gc) =_{df} \forall t < \pi_1(last(tr^\frown s)) \cdot \\ \Big(\neg gc(v_t) \vee \Big(\exists ttr \in (tr^\frown s - pre(last(tr))) \cdot (\pi_1(ttr) = LClosest(tr^\frown s, t) \wedge ttr \rightsquigarrow \neg gd) \Big)\Big), \\ LClosest(tr^\frown s, t) = t_c,\ \textbf{if } t_c \in \pi_1^*(tr^\frown s) \cdot \Big(t_c \leqslant t \wedge (\forall time \in \pi_1^*(tr^\frown s) \cdot (time \leqslant t \rightarrow t_c \geqslant time)) \Big)$$

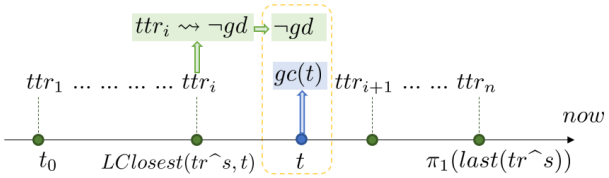


Fig. 1. Demonstration of *Await*

Also, we give an example to show this rule as below. Here, $tr_1 =_{df} \langle(0,\sigma_0,1),(0,\sigma_1,1)\rangle$, $tr_2 =_{df} tr_1{}^\frown\langle(1,\sigma',0),(2,\sigma'',0)\rangle$, $\sigma_0 =_{df} \{x \mapsto 0\}$, $\sigma_1 =_{df} \{x \mapsto 1\}$, $\sigma' =_{df} \{x \mapsto 1, y \mapsto 0\}$ and $\sigma'' =_{df} \{x \mapsto 1, y \mapsto 1\}$.

$$[tr_1 \bullet v = 1.25 \wedge now = 0]\ \dot{v} = 1\ \textbf{until}\ y > 0 \wedge v \geqslant 2.5\ [tr_2 \bullet v = 3.25 \wedge now = 2]$$

(4) Rule 5-4-1 and **Rule 5-4-2** include rules that the guard is a hybrid guard with the form of $gd \vee gc$.

Rule 5-4-1. Cb_HybridGuard2_1

$$\frac{gc(now) \vee (last(tr) \rightsquigarrow gd)}{[tr \bullet p_c]\ R(v,\dot{v})\ \textbf{until}\ gd \vee gc\ [tr \bullet p_c]}$$

Rule 5-4-2. Cb_HybridGuard2_2

$$\frac{p_c[t_0/now] \wedge AwaitTrig(gd \vee gc) \wedge R(v,\dot{v})\ \textbf{during}\ [t_0, now) \rightarrow q_c}{[tr \bullet p_c]\ R(v,\dot{v})\ \textbf{until}\ gd \vee gc\ [tr^\frown s \bullet q_c]}$$

For the hybrid guard $gd \vee gc$, the continuous variable v evolves until the guard gd or gc is satisfied. **Rule 5-4-1** portrays that at least one guard condition is satisfied at the beginning.

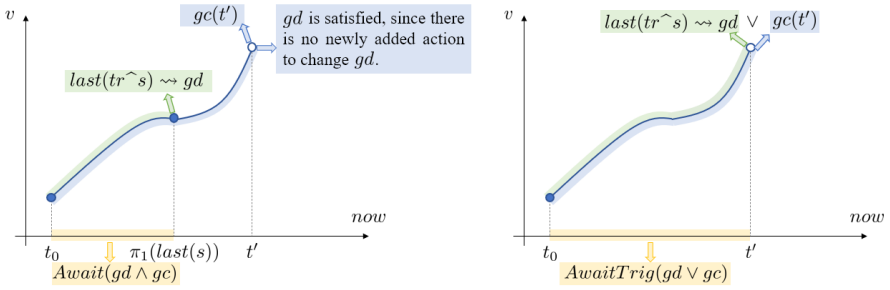
If the current data state cannot meet gd and gc, the program will wait to be triggered. As given in **Rule 5-4-2**, we define $AwaitTrig(gd \vee gc)$ to describe the waiting and the eventual triggering process. The first two lines in the bracket indicate that gc and gd are both unsatisfied before the terminal time t'. The third line informs that it meets gc or gd at the time t'. The demonstrative figure is shown in Fig. 2(b).

$$AwaitTrig(gd \vee gc) =_{df} \exists t' \in (t_0, \infty) \cdot \begin{pmatrix} (\forall t < t' \cdot \neg gc(v_t)) \\ \wedge(\forall ttr \in (tr^\frown s - pre(last(tr))) \cdot (\pi_1(ttr) < t') \rightarrow (ttr \rightsquigarrow \neg gd)) \\ \wedge(gc(v_{t'}) \vee (\pi_1(last(tr^\frown s)) = t' \wedge last(tr^\frown s) \rightsquigarrow gd)) \wedge now = t' \end{pmatrix}$$

According to this rule, we can get:

$$[tr_1 \bullet v = 1.25 \wedge now = 0]\ \dot{v} = 1\ \textbf{until}\ y > 0 \vee v \geqslant 2.5\ [tr_1' \bullet v = 2.5 \wedge now = 1.25].$$

Here, $tr_1 =_{df} \langle(0,\sigma_0,1),(0,\sigma_1,1)\rangle$, $tr_1' =_{df} tr_1{}^\frown\langle(1,\sigma',1)\rangle$, $\sigma_0 =_{df} \{x \mapsto 0\}$, $\sigma_1 =_{df} \{x \mapsto 1\}$ and $\sigma' =_{df} \{x \mapsto 1, y \mapsto 0\}$.



(a) Cb_HybridGuard1

(b) Cb_HybridGuard2

Fig. 2. Cb_HybridGuard

3.3 Proof Rules for Compound Constructs

In this subsection, rules for compound constructs (Sequential Composition, Iteration Construct and Parallel Composition) are enumerated.

Rule 6. Sequential Composition

$$\frac{[tr \bullet p_c]\ P\ [tr_m \bullet p_m],\ [tr_m \bullet p_m]\ Q\ [tr' \bullet q_c]}{[tr \bullet p_c]\ P;Q\ [tr' \bullet q_c]}$$

It requires that the trace and the continuous variables' values in P's post-condition are consistent with those in Q's precondition.

As an example, we assume that $tr_0 =_{df} \langle(0, \sigma_0, 1)\rangle$, $tr_1 =_{df} \langle(0, \sigma_0, 1), (0, \sigma_1, 1)\rangle$, $\sigma_0 =_{df} \{x \mapsto 0\}$ and $\sigma_1 =_{df} \{x \mapsto 1\}$. From

$$[tr_0 \bullet v = 1.25 \wedge now = 0]\ x := x + 1\ [tr_1 \bullet v = 1.25 \wedge now = 0]$$
$$[tr_1 \bullet v = 1.25 \wedge now = 0]\ \dot{v} = 1\ \textbf{until}\ v \geqslant 2.5\ [tr_1 \bullet v = 2.5 \wedge now = 1.25],$$

through **Rule 6**, we have:

$$[tr_0 \bullet v = 1.25 \wedge now = 0]\ x := x + 1; \dot{v} = 1\ \textbf{until}\ v \geqslant 2.5\ [tr_1 \bullet v = 2.5 \wedge now = 1.25].$$

Rule 7. Conditional Choice

$$\frac{\begin{array}{c}[b\&\ (tr \bullet p_c)]\ P\ [tr' \bullet q_c] \\ [\neg b\&\ (tr \bullet p_c)]\ Q\ [tr' \bullet q_c]\end{array}}{[tr \bullet p_c]\ \textbf{if}\ b\ \textbf{then}\ P\ \textbf{else}\ Q\ [tr' \bullet q_c]}$$

The first line of this rule implies that we need to prove $[tr \bullet p_c]\ P\ [tr' \bullet q_c]$, if the Boolean condition b is satisfied in the data state of $last(tr) \bullet p_c$ (i.e., $last(tr) \bullet p_c \rightsquigarrow b$). On the contrary, the second line means that if the data state cannot meet b, then we need to prove $[tr \bullet p_c]\ Q\ [tr' \bullet q_c]$.

We define that $tr_0 =_{df} \langle(0, \sigma_0, 1)\rangle$, $tr_1 =_{df} \langle(0, \sigma_0, 1), (0, \sigma_1, 1)\rangle$, $\sigma_0 =_{df} \{x \mapsto 0\}$ and $\sigma_1 =_{df} \{x \mapsto 1\}$. From this rule, we can prove:

$$[tr_0 \bullet v = 1.25 \wedge now = 0]\ \textbf{if}\ x \geqslant 0\ \textbf{then}\ x := x + 1\ \textbf{else}\ x := x - 1\ [tr_1 \bullet v = 1.25 \wedge now = 0].$$

Rule 8. Iteration Construct

$$\frac{(last(tr) \bullet p_c) \rightsquigarrow I,\ \{I \wedge b\}\ P\ \{I\},\ (last(tr') \bullet q_c) \rightsquigarrow (I \wedge \neg b)}{[tr \bullet p_c]\ \textbf{while}\ b\ \textbf{do}\ P\ [tr' \bullet q_c]}$$

For the iteration construct, we follow the classic definition in Hoare logic and employ loop invariant [4] to prove it. Here, I is the invariant of this iteration and it can be inferred from the precondition. We only need to focus on the last element of tr, since the trace can be generated by the previous programs. The assertion of $\{I \wedge b\}\ P\ \{I\}$ stays the same as that in traditional Hoare Logic. When the program terminates, the data state should satisfy $I \wedge \neg b$ (i.e., $(last(tr') \bullet q_c) \rightsquigarrow (I \wedge \neg b)$).

We take the program below as an example to explain this rule.

$$[tr_0 \bullet v = 1.25 \wedge now = 0]\ \textbf{while}\ x \leqslant 2\ \textbf{do}\ x := x + 1\ [tr_3 \bullet v = 1.25 \wedge now = 0]$$

Here, $tr_0 =_{df} \langle(0, \sigma_0, 1)\rangle$, $tr_3 =_{df} \langle(0, \sigma_0, 1), (0, \sigma_1, 1), (0, \sigma_2, 1), (0, \sigma_3, 1)\rangle$ and $\sigma_i =_{df} \{x \mapsto i\}(i = 0, 1, 2, 3)$. In order to prove this formula, we need to prove the following three premises. We define $x \leqslant 3$ as the invariant I.

- $(last(tr) \bullet p_c) \rightsquigarrow I$ and $(last(tr') \bullet q_c) \rightsquigarrow (I \wedge \neg b)$: They can be deduced with predicate logic directly. The definition of $\rightsquigarrow$ is given in page 5.
- $\{I \wedge b\}\ P\ \{I\}$: To prove this, we transform it to the form of trace, i.e.,

  $$[tr_{Ib}]\ x := x + 1\ [tr_{I'}]$$

  Here, tr_{Ib} stands for the trace whose data state satisfies $I \wedge b$ and $tr_{I'}$ is the terminal trace after executing $x := x + 1$. We define $tr_{Ib} =_{df} \langle(0, \sigma_{Ib}, 1)\rangle$, $PL(\sigma_{Ib}) \rightarrow x \leqslant 2$, $tr_{I'} =_{df} tr_{Ib}{}^\frown\langle(0, \sigma_{I'}, 1)\rangle$ and $\sigma_{I'} = \sigma_{Ib}[x + 1/x]$. With **Rule 3**, $[tr_{Ib}]\ x := x + 1\ [tr_{I'}]$ is proved. Next, **Rule 1** leads to $\{I \wedge b\}\ P\ \{I\}$.

Rule 9. Parallel Composition

$$\frac{\begin{array}{l}[tr \bullet p_c]\ P\ [tr_1 \bullet (q_{c1} \wedge now = t_1)],\\ [tr \bullet p_c]\ Q\ [tr_2 \bullet (q_{c2} \wedge now = t_2)],\\ q_c = q_{c1} \wedge q_{c2},\ tr_{add} \in Merge(tr_1 - tr, tr_2 - tr),\ tr' = tr{}^\frown tr_{add}\end{array}}{[tr \bullet p_c]\ P \parallel Q\ [tr' \bullet q_c[max\{t_1, t_2\}/now]]}$$

In our approach, we employ the trace model to reflect the interaction between the parallel components, so that we can focus on the individual components first and then combine them.

We assume that continuous variables cannot be shared writable in our language. It makes sense, since a continuous variable only has a determined value at one time in the real-world. Therefore, q_{c1} and q_{c2} have no shared writable variables. Thus, we can simply combine q_{c1} and q_{c2} as $q_{c1} \wedge q_{c2}$. Besides, the maximal terminal time of P and Q is the terminal time of $P \parallel Q$. As for the final trace of $P \parallel Q$, we propose a *Merge* function to define the terminal trace of parallel

composition. We define $Merge(trace_1, trace_2)$ as below. Here, $trace_1$, $trace_2$ and $trace_3$ stand for the newly added traces of P, Q and $P \parallel Q$ respectively.

$$Merge(trace_1, trace_2) =_{df}$$
$$\left\{ trace_3 \middle| \begin{array}{c} (\pi_1^*(trace_3) = \pi_1^*(trace_1) = \pi_1^*(trace_2)) \wedge \\ (\pi_2^*(trace_3) = \pi_2^*(trace_1) = \pi_2^*(trace_2)) \wedge \\ (\pi_3^*(trace_3) = \pi_3^*(trace_1) + \pi_3^*(trace_2)) \wedge (2 \notin \pi_3^*(trace_3)) \end{array} \right\}$$

Here, π_i^* is lifted from a single snapshot to a sequence of snapshots. As defined in Section 3, a snapshot in the trace is a tuple (t, σ, μ). The first two conditions imply that the time and data stored in the trace of the parallel composition (i.e., $trace_3$) should be the same as those in both parallel components (i.e., $trace_1$ and $trace_2$). The third one indicates that the actions of parallel components P and Q are also the actions of their parallel composition. The last condition restricts that every snapshot can only be contributed by one parallel component. In the trace model, although the two parallel components can do actions at the same time, snapshots of them need to be added to the trace one by one.

Considering the following parallel program, we illustrate this rule. As an example, from

$$[tr_0 \bullet v = 0 \wedge now = 0]\ x := x + 1; \dot{v} = 1\ \textbf{until}\ v \geqslant 1\ [tr_1' \bullet v = 1 \wedge now = 1],$$
$$[tr_0 \bullet u = 0 \wedge now = 0]\ x := 2; \dot{u} = 2\ \textbf{until}\ u \geqslant 4\ [tr_2' \bullet u = 4 \wedge now = 2],$$

through **Rule 9**, we have:

$$[tr_0 \bullet v = 0 \wedge u = 0 \wedge now = 0]$$
$$(x := x + 1; \dot{v} = 1\ \textbf{until}\ v \geqslant 1) \parallel (x := 2; \dot{u} = 2\ \textbf{until}\ u \geqslant 4)$$
$$[tr' \bullet v = 1 \wedge u = 4 \wedge now = 2]$$

where,

$$\sigma_i =_{df} \{x \mapsto i\} (i = 0, 1, 2, 3),\ tr_0 =_{df} \langle (0, \sigma_0, 1) \rangle,$$
$$tr_1' =_{df} \langle (0, \sigma_0, 1), (0, \sigma_2, 0), (0, \sigma_3, 1) \rangle \vee \langle (0, \sigma_0, 1), (0, \sigma_1, 1), (0, \sigma_2, 0) \rangle,$$
$$tr_2' =_{df} \langle (0, \sigma_0, 1), (0, \sigma_2, 1), (0, \sigma_3, 0) \rangle \vee \langle (0, \sigma_0, 1), (0, \sigma_1, 0), (0, \sigma_2, 1) \rangle,$$
$$tr' =_{df} \langle (0, \sigma_0, 1), (0, \sigma_2, 1), (0, \sigma_3, 1) \rangle \vee \langle (0, \sigma_0, 1), (0, \sigma_1, 1), (0, \sigma_2, 1) \rangle.$$

Here, as introduced before, $tr = tr_a \vee tr_b$ denotes that there are two possible cases of tr, i.e., tr can be equal to tr_a or tr_b. In the above sample parallel program, tr_1' and tr_2' both have two possible traces because they represent two different execution orders of $x := x + 1$ and $x := 2$. As a consequence, the final trace of their parallel composition tr' has two cases through *Merge* function.

4 Case Study

In this section, we present an example of a Battery Management System (BMS) to illustrate the usage of our proof system. We first give the BMS program to demonstrate the syntax of our language. Then, we prove some related properties using our proof system.

4.1 Description of BMS

We employ the process of heat management in BMS as an example to illustrate our language. For simplicity, we assume that the battery works properly if its temperature is between $T_{safemin}$ and $T_{safemax}$. When the vehicle is moving and the temperature does not exceed the threshold value T_{MAX}, the temperature increases linearly. BMS will cool down the battery if the temperature is equal to or greater than $T_{safemax}$. Also, the temperature decreases linearly when BMS is cooling and the temperature does not reach the threshold value T_{MIN}. If the temperature is equal to or less than $T_{safemin}$, BMS will stop cooling.

$$
\begin{array}{l}
BMS =_{df}\ \ Ctrl \parallel Temp;\\
Ctrl =_{df}\ \ \textbf{while}\ DT < 60\ \textbf{do}\\
\left\{\begin{array}{l}
@(caron = 1);\ \dot{t} = 1\ \textbf{until}\ t \geqslant DT + 1;\\
\textbf{if}(\theta \geqslant T_{safemax})\textbf{then}\ coolon := 1;\ \textbf{else}\ coolon := coolon;\\
\textbf{if}(\theta \leqslant T_{safemin})\textbf{then}\ coolon := 0;\ \textbf{else}\ coolon := coolon;\\
DT := t;
\end{array}\right\}\\
Temp =_{df}\ \ \textbf{while}\ DT < 60\ \textbf{do}\\
\left\{\begin{array}{r}
\textbf{if}(coolon == 0)\textbf{then}\ \dot{\theta} = 1\ \textbf{until}(\theta \geqslant T_{MAX} \vee coolon = 1);\\
\textbf{else}\ \dot{\theta} = -2\ \textbf{until}(\theta \leqslant T_{MIN} \vee coolon = 0);
\end{array}\right\}
\end{array}
$$

Here, DT is an auxiliary variable to realize the delay operation. t and θ represents the time and the temperature, and we assume $\theta = 10$ and $t = 0$ at the beginning.. $caron = 1$ means that the car is moving. *coolon* stands for the switch of cooling, BMS starts to cool down the battery if $coolon = 1$. We set $T_{MIN} = 0$, $T_{MAX} = 100$, $T_{safemin} = 10$ and $T_{safemax} = 40$.

4.2 Proof for BMS

We first give an overview of the proof, and then present the proof outline of $Ctrl$ and $Temp$. Finally, we prove the whole program with **Rule 9**.

4.2.1. Overview

The program of BMS should meet that the battery must begin to cool down once the temperature is equal to or greater than $T_{safemax}$, and it stops cooling when the temperature is equal to or lower than $T_{safemin}$. Further, the temperature of the battery is guaranteed to be controlled between $T_{safemin}$ and $T_{safemax}$. Altogether, the program BMS should satisfy the following correctness formula.

$$\{Init\}\ BMS\ \{DI \wedge CI\}$$

where,

$Init =_{df}\ caron = 1 \wedge DT = 0 \wedge coolon = 0 \wedge \theta = 10 \wedge t = 0$,

$DI =_{df}\ (\theta \geqslant T_{safemax} \rightarrow coolon = 1) \wedge (\theta \leqslant T_{safemin} \rightarrow coolon = 0) \wedge DT \leqslant 60 \wedge caron = 1$,

$CI =_{df}\ T_{safemin} \leqslant \theta \leqslant T_{safemax}$.

We first convert this correctness formula to the form of trace model.

$$[tr \bullet p_c \wedge now = 0]\ BMS\ [tr' \bullet q_c \wedge now = 60]$$

Here, since we are only concerned with the last snapshot of tr', we abstract the intermediate snapshots done by the loop body as a sequence of three dots (...). $PL(\sigma') \rightarrow (DI \wedge DT = 60)$ indicates that the terminal data state σ' meets $DI \wedge DT = 60$.

$p_c =_{df} \theta = 10 \wedge t = 0,\ q_c =_{df} 10 \leqslant \theta \leqslant 40 \wedge t = 60,$

$tr =_{df} \langle(0, \sigma, 1)\rangle,\ tr' =_{df} \langle(0, \sigma, 1), ..., (60, \sigma', 1)\rangle,$

$\sigma =_{df} \{caron \mapsto 1, DT \mapsto 0, coolon \mapsto 0\},\ PL(\sigma') \rightarrow (DI \wedge DT = 60).$

4.2.2. Proof for Ctrl

In this part, we want to prove $[tr \bullet p_c \wedge now = 0]\ Ctrl\ [tr_1 \bullet q_{c1} \wedge now = 60]$. The proof outline of $Ctrl$ is given below.

$[tr \bullet p_c \wedge now = 0]$

while $DT < 60$ **do**

$[tr_{Ib} \bullet p_{dt} \wedge now = DT]$...[1]

$@(caron = 1);$

$[tr_{Ib} \bullet p_{dt} \wedge now = DT]$...[2]

$\dot{t} = 1$ **until** $t \geqslant DT + 1;$

$[tr_{Ib} \bullet p_{dt+1} \wedge now = DT + 1]$...[3]

if$(\theta \geqslant T_{safemax})$**then**

$[(\theta \geqslant 40)\&(tr_{Ib} \bullet p_{dt+1} \wedge now = DT + 1)]$...[3.1]

$coolon := 1;$

$[tr_{Ib}{}^\frown\langle(now, \sigma_{c1}, 1)\rangle \bullet p_{dt+1} \wedge now = DT + 1]$...[3.2]

else

$[\neg(\theta \geqslant 40)\&(tr_{Ib} \bullet p_{dt+1} \wedge now = DT + 1)]$...[3.3]

$coolon := coolon;$

$[tr_{Ib}{}^\frown\langle(now, \sigma_{Ib}, 1)\rangle \bullet p_{dt+1} \wedge now = DT + 1]$...[3.4]

$[tr_{Ib}{}^\frown\langle(now, \sigma_{Ib}, 1)\rangle \bullet p_{dt+1} \wedge now = DT + 1]$...[4]

if$(\theta \leqslant T_{safemin})$**then**

$[(\theta \leqslant 10)\&(tr_{Ib1} \bullet p_{dt+1} \wedge now = DT + 1)]$...[4.1]

$coolon := 0;$

$[tr_{Ib1}{}^\frown\langle(now, \sigma_{c2}, 1)\rangle \bullet p_{dt+1} \wedge now = DT + 1]$...[4.2]

else

$[\neg(\theta \leqslant 10)\&(tr_{Ib1} \bullet p_{dt+1} \wedge now = DT + 1)]$...[4.3]

$coolon := coolon;$

$[tr_{Ib1}{}^\frown\langle(now, \sigma_{Ib}, 1)\rangle \bullet p_{dt+1} \wedge now = DT + 1]$...[4.4]

$[tr_{Ib1}{}^\frown\langle(now, \sigma_{Ib}, 1)\rangle \bullet p_{dt+1} \wedge now = DT + 1]$...[5]

$DT := t;$

$[tr_{Ib2}{}^\frown\langle(now, \sigma_t, 1)\rangle \bullet p_{dt+1} \wedge now = DT + 1]$...[6]

$[tr_{I'} \bullet p_{dt+1} \wedge now = DT + 1]$...[7]

$[tr_1 \bullet q_{c1} \wedge now = 60]$

Also, some notations are defined as below.

$p_{dt} =_{df} t = DT,\ p_{dt+1} =_{df} t = DT + 1,\ q_{c1} =_{df} true,$

$tr_1 =_{df} tr',\ tr_{Ib} =_{df} \langle(now, \sigma_{Ib}, 1)\rangle,\ tr_{Ib1} =_{df} tr_{Ib}{}^\frown\langle(now, \sigma_{Ib}, 1)\rangle,$

$tr_{Ib2} =_{df} tr_{Ib1}{}^\frown\langle(now, \sigma_{Ib}, 1)\rangle,\ tr_{I'} =_{df} tr_{Ib2}{}^\frown\langle(now, \sigma_I, 1)\rangle,$

$PL(\sigma_I) \rightarrow DI,\ PL(\sigma_{Ib}) \rightarrow (DI \wedge DT < 60),$

$\sigma_{c1} =_{df} \sigma_{Ib}[1/coolon],\ \sigma_{c2} =_{df} \sigma_{Ib}[0/coolon],\ \sigma_t =_{df} \sigma_{Ib}[t/DT].$

4.2.3. Proof for Temp

In this part, we want to prove $[tr \bullet p_c \wedge now = 0]\ Temp\ [tr_2 \bullet q_{c2} \wedge now = 60]$. For gaining it, we present the proof outline of $Temp$ as below.

$[tr \bullet p_c \wedge now = 0]$

while $DT < 60$ **do**

$\left\{\begin{array}{l} [tr_{Ib'} \bullet \theta = \theta_i \wedge now = t_i]...[1] \\ \textbf{if}(coolon == 0)\textbf{then} \\ [(coolon == 0)\&(tr_{Ib'} \bullet \theta = \theta_i \wedge now = t_i)]...[1.1] \\ \quad \dot{\theta} = 1\ \textbf{until}(\theta \geqslant T_{MAX} \vee coolon = 1); \\ [tr_{Ib'}{}^\frown\langle(now, \sigma_{c1}, 0)\rangle \bullet \theta = 40 \wedge now = t_i + 40 - \theta_I]...[1.2] \\ \textbf{else} \\ [\neg(coolon == 0)\&(tr_{Ib'} \bullet \theta = \theta_i \wedge now = t_i)]...[1.3] \\ \quad \dot{\theta} = -2\ \textbf{until}(\theta \leqslant T_{MIN} \vee coolon = 0); \\ [tr_{Ib'}{}^\frown\langle(now, \sigma_{c2}, 0)\rangle \bullet \theta = 10 \wedge now = t_i + \theta_i - 10]...[1.4] \\ [tr_{I''} \bullet \theta_I \wedge now = (t_i + 40 - \theta_i) \vee (t_i + \theta_i - 10)]...[2] \end{array}\right\}$

$[tr_2 \bullet q_{c2} \wedge now = 60]$

where,

$$10 \leqslant \theta_i \leqslant 40,\ q_{c2} =_{df} CI,\ \theta_I =_{df} CI,\ tr_{Ib'} =_{df} \langle(now, \sigma_{Ib}, 0)\rangle,$$
$$tr_{I''} =_{df} tr_{Ib'}{}^\frown\langle(now, \sigma_I, 0)\rangle,\ tr_2 =_{df} \langle(0, \sigma, 1), ..., (60, \sigma', 0)\rangle.$$

4.2.4. Parallel Composition of Ctrl and Temp

By means of **Rule 9**, we get:

$$\frac{\begin{array}{l}[tr \bullet p_c \wedge now = 0]\ Ctrl\ [tr_1 \bullet q_{c1} \wedge now = 60], \\ [tr \bullet p_c \wedge now = 0]\ Temp\ [tr_2 \bullet q_{c2} \wedge now = 60], \\ q_c = q_{c1} \wedge q_{c2},\ tr_{add} \in Merge(tr_1 - tr, tr_2 - tr),\ tr' = tr{}^\frown tr_{add}\end{array}}{[tr \bullet p_c \wedge now = 0]\ Ctrl \parallel Temp\ [tr' \bullet q_c \wedge now = 60]}$$

In $Ctrl \parallel Temp$, we note that the actions of $Temp$ are all continuous. As a result, the actions recorded in tr' are exactly the same as those in tr_1 that were done by $Ctrl$. Thus, snapshots in $tr_1 - tr$ and $tr_2 - tr$ are structurally similar and vary only in their values of μ. Then, the satisfaction of $tr_{add} \in Merge(tr_1 - tr, tr_2 - tr)$ is obvious. We employ **Rule 1**, and obtain the final desirable result, i.e., $\{Init\}\ BMS\ \{DI \wedge CI\}$.

5 Conclusion and Future Work

In this paper, we have presented a proof system for the language which is proposed to model cyber-physical systems based on our previous work [1, 8]. We extended the triple in classical Hoare Logic $\{p\}\ S\ \{q\}$ to $[tr \bullet p_c]\ S\ [tr' \bullet q_c]$. In this specification, tr and tr' are introduced to pave the way for the proof of parallel programs with shared variables. Considering that trace can only record

the values of discrete variables, we also appended a precondition p_c and a post-condition q_c to indicate the states of continuous variables. *now*, the global clock variable, is added to catch the feature of real-time in CPS as well. Our proof system is mainly supported by the trace model, it can not only realize compositional proof, but also link the proof system with denotational semantics [8]. For an intuitive demonstration of this proof system's usage, we also provided the example of a battery management system.

In the future, considering that the traditional approach of building the link between the proof system and semantics is mainly based on operational semantics, we will investigate the link between our proof system and our denotational semantics [8] in detail.

Acknowledgements. This work was partly supported by the National Key Research and Development Program of China (Grant No. 2018YFB2101300), the National Natural Science Foundation of China (Grant Nos. 61872145, 62032024), Shanghai Trusted Industry Internet Software Collaborative Innovation Center, and the Dean's Fund of Shanghai Key Laboratory of Trustworthy Computing (East China Normal University).

References

1. Banach, R., Zhu, H.: Language evolution and healthiness for critical cyber-physical systems. J. Softw. Evol. Process. **33**(9) (2021)
2. He, J., Li, Q.: A hybrid relational modelling language. In: Concurrency, Security, and Puzzles. Lecture Notes in Computer Science, vol. 10160, pp. 124–143. Springer (2017)
3. Henzinger, T.A.: The theory of hybrid automata. In: LICS. pp. 278–292. IEEE Computer Society (1996)
4. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
5. Jones, C.B.: Accommodating interference in the formal design of concurrent object-based programs. Formal Methods Syst. Des. **8**(2), 105–122 (1996)
6. Lanotte, R., Merro, M., Tini, S.: A probabilistic calculus of cyber-physical systems. Inf. Comput. **279**, 104618 (2021)
7. Lee, E.A.: Cyber physical systems: Design challenges. In: ISORC. pp. 363–369. IEEE Computer Society (2008)
8. Li, R., Zhu, H., Banach, R.: Denotational and algebraic semantics for cyber-physical systems. In: ICECCS. pp. 123–132. IEEE (2022)
9. Liu, J., Lv, J., Quan, Z., Zhan, N., Zhao, H., Zhou, C., Zou, L.: A calculus for hybrid CSP. In: APLAS. Lecture Notes in Computer Science, vol. 6461, pp. 1–15. Springer (2010)
10. Lunel, S., Mitsch, S., Boyer, B., Talpin, J.: Parallel composition and modular verification of computer controlled systems in differential dynamic logic. In: FM. Lecture Notes in Computer Science, vol. 11800, pp. 354–370. Springer (2019)
11. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Informatica **6**, 319–340 (1976)
12. Zhou, C., Wang, J., Ravn, A.P.: A formal description of hybrid systems. In: Hybrid Systems. Lecture Notes in Computer Science, vol. 1066, pp. 511–530. Springer (1995)