An Algebraic Approach to Simulation and Verification for Cyber-Physical Systems with Shared-Variable Concurrency

Ran Li[a,b,c], Huibiao Zhu[d,*] and Richard Banach[e]

[a] School of Software, Nanjing University of Information Science and Technology, Nanjing, China
[b] Automatic Software Generation and Intelligent Service Key Laboratory of Sichuan Province, Chengdu University of Information Technology, Chengdu, China
[c] Jiangsu Province Engineering Research Center of Advanced Computing and Intelligent Services, Nanjing University of Information Science and Technology, Nanjing, China
[d] Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China
[e] Department of Computer Science, University of Manchester, Manchester, UK

ARTICLE INFO

Keywords:
Cyber-Physical Systems (CPS)
Shared Variables
Algebraic Semantics
Simulation
Verification
Real-Time Maude

ABSTRACT

Cyber-Physical systems (CPS), containing discrete behaviors of the cyber and continuous behaviors of the physical, have gained wide applications in many fields. Since CPS subsume the intersection of cyber systems and physical processes, the traditional modeling languages which merely include discrete variables are no longer applicable to CPS. Accordingly, a shared variable language called $CPSL^{sc}$ was proposed to specify CPS. In this paper, we elaborate the algebraic semantics for this language, so that every program of $CPSL^{sc}$ can be converted into a unified form called guarded choice form and the sequentialization of parallel programs is achieved. Additionally, we formalize the algebraic semantics in the rewriting engine Real-Time Maude. With the algebraic laws constructed, for every program specified with $CPSL^{sc}$, we can simulate its execution step by step. Furthermore, automatic transformation and execution are attained. As a consequence, if the program and its initial data state are provided, the corresponding trace of data states during execution can be generated. In the light of the generated trace, automatic verification can be carried out as well.

1. Introduction

Cyber-Physical systems (CPS) are multidimensional complex systems that integrate discrete behaviors and continuous behaviors. In CPS, computer programs can influence physical behaviors, and vice versa. CPS has covered a wide range of application areas, including healthcare equipment, intelligent traffic control and environmental monitoring, etc.

Meanwhile, the interaction between the cyber and the physical makes the traditional modeling languages defined for purely discrete systems unsuitable for CPS. Therefore, some specification languages are proposed to describe and model CPS, such as Hybrid CSP [1], HRML [2] and so on. We proposed a language whose parallel mechanism is based on shared variables in our previous work [3], and we name it $CPSL^{sc}$ (Cyber-Physical Systems Modeling Language with Shared-Variable Concurrency) in this paper. As for this shared variable language, we proposed its formal semantics [4], developed its proof system [5], and presented an implementation of the transformation from this language to the automata in SpaceEx [6, 7].

This paper extends our work published at ICECCS 2022 [4]. In our previous work, we introduced five types of guarded choices and explored the algebraic semantics of our language. Our previous work [4] mainly focuses on the theoretical view of the algebraic semantics. Now, compared with the previous work, we elaborate the algebraic semantics and apply the algebraic semantics into practice. This paper aims to carry out the simulation and verification of CPS from the view of formal semantics, particularly the algebraic semantics. The main contributions of this paper are presented in Figure 1.

- **From Theoretical View:** We elaborate the algebraic semantics by redefining the guarded component and guarded choice of continuous behaviors. With the updated definitions, we propose the algebraic semantics. Then, based on the proposed semantics, every program described by $CPSL^{sc}$ can be transformed into the unified

*Corresponding author: hbzhu@sei.ecnu.edu.cn
ORCID(s): 0000-0002-3492-1591 (R. Li); 0000-0002-0214-8565 (H. Zhu)

form (i.e., guarded choice form), and parallel programs with discrete and continuous behaviors can be sequentialized consequently.

- **From Practical View:** We add the mechanization of the algebraic semantics in the rewriting engine Real-Time Maude [8, 9], simulation and verification of CPS can be conducted accordingly. Additionally, to better understand the usage of the semantics and its implementation in Maude, we employ the program of a battery management system (BMS) as a case study in this paper.

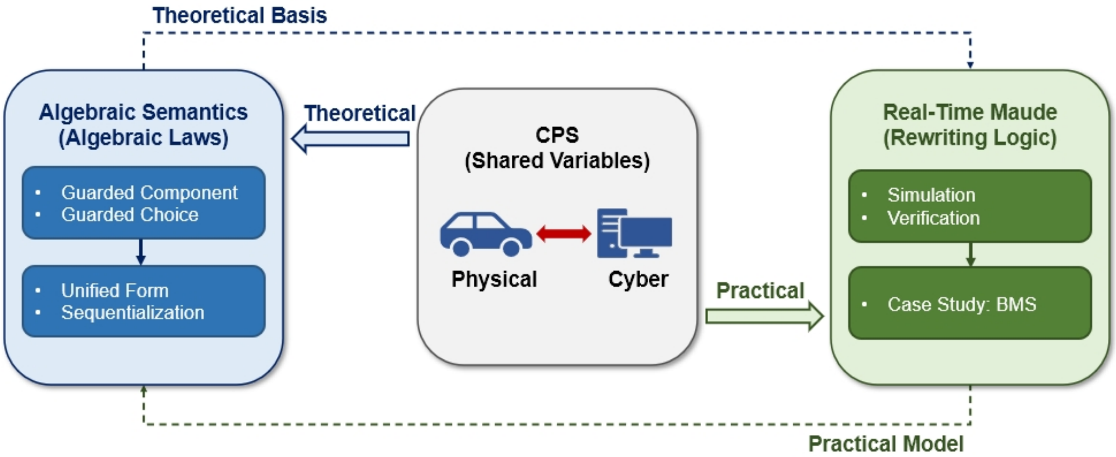

Figure 1: Technology roadmap

More specifically, in this paper, the updated forms of guarded component and guarded choice for continuous behaviors consider the continuous behavior as a whole rather than splitting it. The new form is on the one hand closer to the continuous nature of continuous behaviors, and on the other hand, has a more concise form of expression. Based on the updated guarded components and guarded choices, the elaborated algebraic semantics is presented. Subsequently, any program in $CPSL^{sc}$ can be expressed as the guarded choice form. Actually, the core of the guarded choice form in our algebraic semantics is deriving the first action from the program. Therefore, for a given program (sequential or parallel), we can get the first action of the program step by step, that is, sequentialization is achieved.

Then, we consider how to apply the proposed formal semantics to simulation and verification. Program properties can be expressed as algebraic laws (equations usually), which can be verified using the formalized semantics. As mentioned in [10], "Algebra is well-suited for direct use by engineers in symbolic calculation of parameters and the structure of an optimal design. Algebraic proof by term rewriting is a most promising way in which computers can assist in the process of reliable design." As a result, this paper aims to build a bridge between the algebraic semantics and its applications (i.e., simulation and verification) through term rewriting.

Real-Time Maude [8, 9] is the extension of the rewriting engine Maude [11], its theoretical basis is rewriting logic [12], and it supports formal specification, simulation and analysis of real-time and hybrid systems. Therefore, we implement the proposed algebraic laws and then conduct simulation and verification in Real-Time Maude. Through the mechanical implementation, for the input program attached with its initial state, Maude can automatically convert the program to the guarded choice form and then execute it. The final result returned by Maude is a trace of data states. These data states are recorded during the execution of the program, and they reflect the execution of the program. Moreover, inspired by the definition of "*Monitor*" in the runtime verification [13] and the gist of our algebraic semantics (i.e., obtaining the first action of a given program), we conduct the verification in Maude from a "*trace*" perspective.

The rest of this paper is organized as follows. In Section 2, we recall the syntax of $CPSL^{sc}$ and sketch the rewriting engine Real-Time Maude. Based on the introduction of algebraic guards, guarded components and guarded choices, the algebraic semantics of $CPSL^{sc}$ is explored in Section 3. Section 4 is about the mechanical implementation of algebraic semantics in Real-Time Maude. The case study of a battery management system is carried out in Section 5. Related work is discussed in Section 6. Finally, we conclude our work and propose some future work in Section 7.

Table 1
Syntax of $CPSL^{sc}$

Process	$P, P' ::= S$	(Sequential Process)
	$\mid P \parallel P'$	(Parallel Composition)
Sequential process	$S, S' ::= Db$	(Discrete Behavior)
	$\mid Cb$	(Continuous Behavior)
	$\mid S; S'$	(Sequential Composition)
	$\mid$ **if** b **then** S **else** S'	(Conditional Construct)
	$\mid$ **while** b **do** S	(Iteration Construct)
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	$gc ::= true \mid v = e \mid v < e \mid v > e \mid gc \vee gc \mid gc \wedge gc \mid \neg gc$	

2. Background

In this section, we first recall the syntax of $CPSL^{sc}$. Then, the rewriting engine Real-Time Maude is introduced briefly.

2.1. Syntax

The syntax of $CPSL^{sc}$ is summarized in Table 1. It was proposed in our previous work [3], and we elaborated it by detailing the guard conditions of the continuous behaviors in [4]. Here, x and v stand for discrete and continuous variables respectively. e is an expression that contains discrete or continuous variables. b represents a Boolean condition. In $CPSL^{sc}$, a process includes discrete behaviors Db, continuous behaviors Cb and several compositions and constructs of them.

- **Db:** There are two discrete behaviors in $CPSL^{sc}$, i.e., discrete assignment and discrete event guard.
  - $x := e$ is a discrete assignment. 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. gd is a discrete guard. $@gd$ is triggered if gd is satisfied. Otherwise, the process waits for the environment to trigger gd.
- **Cb:** To describe the continuous behaviors in CPS, we introduce differential relation in $CPSL^{sc}$.
  - $R(v, \dot{v})$ **until** g portrays the continuous behaviors. $R(v, \dot{v})$ is a differential relation which defines the dynamics of the continuous variable v. The continuous variable v evolves as the differential relation specifies until the guard condition g is triggered. In $CPSL^{sc}$, four kinds of guard condition g are allowed, i.e., discrete guard gd, continuous guard gc, mixed guards $gd \wedge gc$ and $gd \vee gc$.
- **Sequential Process:** A sequential process can be comprised of the above commands.
  - $S; S'$ is sequential composition. If the process S terminates successfully, the process S' is executed.
  - **if** b **then** S **else** S' is a conditional construct. If the Boolean condition b is true, then S is executed. Otherwise, S' is executed.
  - **while** b **do** S is an iteration construct. S keeps running repeatedly until the Boolean condition b does not hold.
- **Parallel Composition:** Additionally, a process can also be in the form of parallel composition.
  - $P \parallel P'$ is parallel composition. It indicates P executes in parallel with P'. The parallel mechanism of our language is based on shared variables. In $CPSL^{sc}$, parallel composition occurs only at the outmost level, i.e., $(P||Q); (P'||Q')$ is illegal.

2.2. Real-Time Maude

Maude [11] is based on rewriting logic [12], and Real-Time Maude [8, 9] is the extension of Maude. Real-Time Maude is a language and tool which supports formal specification, simulation and analysis of real-time and hybrid systems.

First, we briefly introduce the syntax of Maude. Maude includes two kinds of modules: functional modules and system modules. We apply the keywords `fmod - endfm` to define a functional module, and the basic data types (i.e., sorts, operators, variables and equations) of a system can be declared in it. A system module is declared with the keyword `mod - endm`, and it can also contain rewrite rules which are defined to specify the dynamic part of a system. Here, we only list the syntax that appears in this paper.

- `sort` and `sorts`: `sort` is used to declare one sort, or we can use `sorts` to declare many sorts. A sort *A* can be defined as a subsort of a sort *B* by `subsort A < B`.
- `op`: An operator is declared with the keyword `op` in the form of `op f : s1 ... sn -> s [Operator Attributes]`. The operators can have some equational attributes, such as `id` (the operator has a certain identity element), `assoc` (the operator is associative) and `comm` (the operator is commutative). Moreover, the `ctor` attribute means that the operator is defined as a constructor which is used to build data elements and has no computational meaning. Also, precedence values can be equipped with operator declarations, where a lower value indicates a tighter binding.
- `var` and `vars`: Variables in Maude are defined with the keyword `var` with the sort following behind the name.
- `eq` and `ceq`: Unconditional equations and conditional equations are declared using the keywords `eq` and `ceq`.
- `rl` and `crl`: The keywords `rl` and `crl` are introduced to declare unconditional rewrite rules and conditional rewrite rules.

Different from the modules in the ordinary Maude, Real-Time Maude declares a data sort *Time* to specify the time domain and all modules are declared as timed modules with the keyword `tmod - endtm`. Further, timed module automatically imports the sorts *System* and *GlobalSystem* and the operator {_} as a skeleton time domain. For the rewrite rules, Real-Time Maude includes two types of rules.

- Instantaneous rewrite rules: They stand for ordinary rewrite rules used to model instantaneous behaviors, and we assume they take zero time.
- Tick rewrite rules: To model the elapse of time in a real-time system, tick rewrite rules are introduced to Real-Time Maude. It has the form of `rl [l]:{t} => {t'} in time` τ or `crl [l]:{t} => {t'} in time` τ `if cond`. The former is unconditional, and the latter is conditional. τ is a term of sort *Time* representing the duration of the tick rewrite rule, `t` and `t'` can be considered as the initial and terminal value respectively.

3. Algebraic Semantics

In this section, we investigate the algebraic semantics of $CPSL^{sc}$. First, we gently introduce notations of algebraic guards, guarded components and guarded choices defined to support our expansion laws. Based on them, we propose algebraic laws for basic statements and parallel composition respectively. Finally, we conclude the usage of the proposed algebraic semantics.

3.1. Notation

To support the parallel expansion laws, algebraic guards, guarded components and guarded choices are introduced.

3.1.1. Algebraic guards

According to the syntax of $CPSL^{sc}$, there are three kinds of basic behaviors, including discrete assignment, discrete event guard and continuous behavior. Hence, we set three types of algebraic guards, and the algebraic guards can be seen as the first action of a given program.

- **Assignment guard:** $b\&@(x := e)$, where b is a Boolean condition

- **Event guard:** $EvtG(@(gd))$
- **Continuous behavior (cb) guard:** $CbG(Cb)$, where Cb is a continuous behavior

Note that we set cb guard as the form of $CbG(Cb)$ instead of Cb^1 in our previous work [4]. Previously, we formalize the cb guard by splitting Cb into several parts, and the behavior of Cb is the sequential composition of these parts. Now, we consider Cb as a whole, i.e., Cb is supposed to evaluate until its boundary g is reached. The elaborated form is not only closer to the continuous nature of continuous behaviors, but also provides more concise form of expression. Based on this change, the subsequent guarded component and guarded choices regarding Cb are also elaborated correspondingly.

3.1.2. Guarded components

Then, $h \to P$ is a guarded component if h is an algebraic guard and P stands for the successive process. Following the above definitions of algebraic guards, there are three types of guarded components.

- **Assignment guarded component:** $b\&@(x := e) \to P$
- **Event guarded component:** $EvtG(@(gd)) \to P$
- **Continuous behavior (cb) guarded component:** $CbG(Cb) \to P$

3.1.3. Guarded choices

Further, $[]\{h_1 \to P_1, ..., h_n \to P_n\}$ is a guarded choice if every element in $\{h_1 \to P_1, ..., h_n \to P_n\}$ is a guarded component. Here, $[]\{h_1 \to P_1, ..., h_n \to P_n\}$ is another written form of $[]h_1 \to P_1[]...[]h_n \to P_n$. Based on the introduced guarded components, we propose five types of guarded choices as below. The first three contain a set of one type of guarded components and the last two are hybrid guarded choices.

- **Assignment guarded choice:** $[]_{i\in I}\{b_i\&@(x_i := e_i) \to P_i\}$
  It contains a set of assignment guarded components. If b_i is satisfied, $x_i := e_i$ can be executed and then the corresponding program P_i will be performed. Moreover, the Boolean conditions $\{b_1, ..., b_n\}$ of the assignment guard components should satisfy $\vee_{i\in I} b_i = true$. This restriction is to ensure that at least one assignment can be executed. This understanding is similar to the if-else statement in the traditional languages.

- **Event guarded choice:** $[]_{i\in I}\{EvtG(@(gd_i)) \to P_i\}$
  It is composed of several event guarded components, and it waits for any guards to be triggered. If $@(gd_i)$ is triggered, the following program P_i will be executed. It is worth noting that we set this guarded choice is urgent, which means that if the current data state satisfies gd_i, then $EvtG(@(gd_i))$ will be scheduled immediately.

- **Cb guarded choice:** $[]\{CbG(Cb) \to P\}$
  It contains cb guarded component. Cb can be executed, indicating that the continuous behavior is performing and its subsequent behavior is described as the program P.

- **Event&assignment hybrid guarded choice:** $[]_{i\in I}\{EvtG(@(gd_i)) \to P_i\} \rhd []_{j\in J}\{b_j\&@(x_j := e_j) \to Q_j\}$
  It is composed of event guarded components and assignment guarded components. If b_j is *true*, $x_j := e_j$ can be selected to execute and then Q_j will be executed. Meanwhile, the system is waiting for the event guard $@(gd_i)$ to be triggered. If the current state meets $@(gd_i)$, the following program P_i will be executed. Likewise, to ensure the immediate trigger of $@(gd_j)$, $EvtG(@(gd_j))$ is considered to have a higher priority than the assignment guards. We adopt the operator $\rhd$ to explicate the precedence, where the left side of $\rhd$ (i.e., event guards) takes precedence over the right side (i.e., assignment guards).

- **Event&cb hybrid guarded choice:** $[]_{i\in I}\{EvtG(@(gd_i)) \to P_i\} \rhd []\{CbG(Cb) \to Q\}$
  It is composed of event guarded components and cb guarded component. The event guard $@(gd_i)$ is waiting to be triggered, and if it can be triggered at present, P_i will be executed. Additionally, the continuous behavior Cb is performing if none of the guards can be triggered now, that is, event guards have higher priority than cb guards. Similarly, the priority is explicated by the operator $\rhd$.

Table 2
Parallel composition of guard choices

	Assignment	**Event**	**Cb**	**Event&Assignment**	**Event&Cb**
Assignment	(par-1-1)	(par-1-2)	(par-1-3)	(par-1-4)	(par-1-5)
Event		(par-2-2)	(par-2-3)	(par-2-4)	(par-2-5)
Cb			(par-3-3)	(par-3-4)	(par-3-5)
Event&Assignment				(par-4-4)	(par-4-5)
Event&Cb					(par-5-5)

Whereas, considering that an assignment is instantaneous and a continuous behavior is not, if an assignment guarded component and a continuous behavior guarded component appear in the same set of a guarded choice, the continuous behavior Cb will never have a chance to be scheduled first. Thus, there is no assignment & cb hybrid guarded choice.

3.2. Algebraic laws

In this subsection, according to the above guarded components and guarded choices, we now present the algebraic laws for $CPSL^{sc}$. Through these algebraic laws, we can transform every program of $CPSL^{sc}$ into the guarded choice form, and the parallel programs with discrete and continuous behaviors can be sequentialized consequently.

3.2.1. Algebraic laws for basic statements

First, we study the laws for basic statements as below. These laws aim to transform programs into the guarded choice form. Here, ε represents an empty program, and $skip$ stands for a special assignment $x := x$.

- **(assign-1)** $x := e \ = [\!]\{true\&@(x := e) \rightarrow \varepsilon\}$
- **(guard-1)** $@(gd) \ = [\!]\{EvtG(@(gd)) \rightarrow \varepsilon\}$
- **(cb-1)** $R(v, \dot{v})$ **until** $g \ = [\!]\{CbG(R(v, \dot{v})$ **until** $g) \rightarrow \varepsilon\}$
- **(cond-1) if** b **then** P **else** $Q \ = [\!]\{b\&@(skip) \rightarrow P\}[\!][\!]\{\neg b\&@(skip) \rightarrow Q\}$
- **(iter-1) while** b **do** $P \ = [\!]\{b\&@(skip) \rightarrow (P;$ **while** b **do** $P)\}[\!][\!]\{\neg b\&@(skip) \rightarrow \varepsilon\}$
- **(seq-1)** $(P; Q); R = P; (Q; R)$
- **(seq-2)** If $P = [\!]\{g_1 \rightarrow P_1, ..., g_n \rightarrow P_n\}$, then $P; Q = [\!]\{g_1 \rightarrow (P_1; Q), ..., g_n \rightarrow (P_n; Q)\}$.

3.2.2. Algebraic laws for parallel composition

Then, we look into the algebraic laws for parallel composition. As summarized in **(par-0-1)** and **(par-0-2)**, parallel composition is symmetric and associative. **(par-0-3)** indicates if one of the parallel components is an empty program ε, then the whole parallel composition is left with its partner P.

- **(par-0-1)** $P \parallel Q = Q \parallel P$
- **(par-0-2)** $(P \parallel Q) \parallel R = P \parallel (Q \parallel R)$
- **(par-0-3)** $P \parallel \varepsilon = P$

As mentioned previously, through the above laws for basic statements, programs in $CPSL^{sc}$ can be translated into the guarded choice form. Thus, for the algebraic laws for parallel composition, we only need to study the parallel composition of their guarded choices. There are five types of guarded choices, and there should be 25 parallel expansion laws. As shown in Table 2, we only need to list 15 laws thanks to the symmetry of the parallel composition.

We propose the parallel expansion laws of $CPSL^{sc}$ in Table 3. The detailed explanations of these algebraic laws are listed below.

Table 3
Algebraic laws for parallel composition

(par)	P	Q	$P \parallel Q$
par-1-1	$[]_{i\in I}\{b_i\&@(x_i := e_i) \to P_i\}$	$[]_{j\in J}\{b_j\&@(x_j := e_j) \to Q_j\}$	$[]_{i\in I}\{b_i\&@(x_i := e_i) \to (P_i \parallel Q)\}$ $[][]_{j\in J}\{b_j\&@(x_j := e_j) \to (P \parallel Q_j)\}$
par-1-2	$[]_{i\in I}\{b_i\&@(x_i := e_i) \to P_i\}$	$[]_{j\in J}\{EvtG(@(\xi_j)) \to Q_j\}$	$[]_{j\in J}\{EvtG(@(\xi_j)) \to (P \parallel Q_j)\}$ $\triangleright[]_{i\in I}\{b_i\&@(x_i := e_i) \to (P_i \parallel Q)\}$
par-1-3	$[]_{i\in I}\{b_i\&@(x_i := e_i) \to P_i\}$	$[]\{CbG(Cb) \to S\}$	$[]_{i\in I}\{b_i\&@(x_i := e_i) \to (P_i \parallel Q)\}$
par-1-4	$[]_{i\in I}\{b_i\&@(x_i := e_i) \to P_i\}$	$[]_{k\in K}\{EvtG(@(\xi_k)) \to T_k\}$ $\triangleright[]_{j\in J}\{b_j\&@(x_j := e_j) \to S_j\}$	$[]_{k\in K}\{EvtG(@(\xi_k)) \to (P \parallel T_k)\}$ $\triangleright[]_{i\in I}\{b_i\&@(x_i := e_i) \to (P_i \parallel Q)\}$ $[][]_{j\in J}\{b_j\&@(x_j := e_j) \to (P \parallel S_j)\}$
par-1-5	$[]_{i\in I}\{b_i\&@(x_i := e_i) \to P_i\}$	$[]_{j\in J}\{EvtG(@(\xi_j)) \to S_j\}$ $\triangleright[]\{CbG(Cb) \to T\}$	$[]_{j\in J}\{EvtG(@(\xi_j)) \to (P \parallel S_j)\}$ $\triangleright[]_{i\in I}\{b_i\&@(x_i := e_i) \to (P_i \parallel Q)\}$
par-2-2	$[]_{i\in I}\{EvtG(@(\xi_i)) \to P_i\}$	$[]_{j\in J}\{EvtG(@(\eta_j)) \to Q_j\}$	$[]_{i\in I}\{EvtG(@(\xi_i)) \to (P_i \parallel Q)\}$ $[][]_{j\in J}\{EvtG(@(\eta_j)) \to (P \parallel Q_j)\}$
par-2-3	$[]_{i\in I}\{EvtG(@(\xi_i)) \to P_i\}$	$[]\{CbG(Cb) \to S\}$	$[]_{i\in I}\{EvtG(@(\xi_i)) \to (P_i \parallel Q)\}$ $\triangleright[]\{CbG(Cb) \to (P \parallel S)\}$
par-2-4	$[]_{i\in I}\{EvtG(@(\xi_i)) \to P_i\}$	$[]_{k\in K}\{EvtG(@(\eta_k)) \to T_k\}$ $\triangleright[]_{j\in J}\{b_j\&@(x_j := e_j) \to S_j\}$	$[]_{i\in I}\{EvtG(@(\xi_i)) \to (P_i \parallel Q)\}$ $[][]_{k\in K}\{EvtG(@(\eta_k)) \to (P \parallel T_k)\}$ $\triangleright[]_{j\in J}\{b_j\&@(x_j := e_j) \to (P \parallel S_j)\}$
par-2-5	$[]_{i\in I}\{EvtG(@(\xi_i)) \to P_i\}$	$[]_{j\in J}\{EvtG(@(\eta_j)) \to S_j\}$ $\triangleright[]\{CbG(Cb) \to T\}$	$[]_{i\in I}\{EvtG(@(\xi_i)) \to (P_i \parallel Q)\}$ $[][]_{j\in J}\{EvtG(@(\eta_j)) \to (P \parallel S_j)\}$ $\triangleright[]\{CbG(Cb) \to (P \parallel T)\}$
par-3-3	$[]\{CbG(Cb_s) \to S\}$	$[]\{CbG(Cb_t) \to T\}$	$[]\{CbG(Cb_{st}) \to ((Re(Cb_{st}, Cb_s); S)\vert\vert(Re(Cb_{st}, Cb_t); T))\}$
par-3-4	$[]\{CbG(Cb) \to N\}$	$[]_{k\in K}\{EvtG(@(\xi_k)) \to T_k\}$ $\triangleright[]_{j\in J}\{b_j\&@(x_j := e_j) \to S_j\}$	$[]_{k\in K}\{EvtG(@(\xi_k)) \to (P \parallel T_k)\}$ $\triangleright[]_{j\in J}\{b_j\&@(x_j := e_j) \to (P \parallel S_j)\}$
par-3-5	$[]\{CbG(Cb_s) \to N\}$	$[]_{j\in J}\{EvtG(@(\eta_j)) \to S_j\}$ $\triangleright[]\{CbG(Cb_t) \to T\}$	$[]_{j\in J}\{EvtG(@(\eta_j)) \to (P \parallel S_j)\}$ $\triangleright[]\{CbG(Cb_{st}) \to ((Re(Cb_{st}, Cb_s); N)\vert\vert(Re(Cb_{st}, Cb_t); T))\}$
par-4-4	$[]_{j\in J}\{EvtG(@(\xi_j)) \to T_j\}$ $\triangleright[]_{i\in I}\{b_i\&@(x_i := e_i) \to S_i\}$	$[]_{n\in N}\{EvtG(@(\eta_n)) \to R_n\}$ $\triangleright[]_{k\in K}\{b_k\&@(x_k := e_k) \to M_k\}$	$[]_{j\in J}\{EvtG(@(\xi_j)) \to (T_j \parallel Q)\}$ $[][]_{n\in N}\{EvtG(@(\eta_n)) \to (P \parallel R_n)\}$ $\triangleright[]_{i\in I}\{b_i\&@(x_i := e_i)) \to (S_i \parallel Q)\}$ $[][]_{k\in K}\{b_k\&@(x_k := e_k)) \to (P \parallel M_k)\}$
par-4-5	$[]_{j\in J}\{EvtG(@(\xi_j)) \to T_j\}$ $\triangleright[]_{i\in I}\{b_i\&@(x_i := e_i) \to S_i\}$	$[]_{k\in K}\{EvtG(@(\eta_k)) \to M_k\}$ $\triangleright[]\{CbG(Cb) \to N\}$	$[]_{j\in J}\{EvtG(@(\xi_j)) \to (T_j \parallel Q)\}$ $[][]_{k\in K}\{EvtG(@(\eta_k)) \to (P \parallel M_k)\}$ $\triangleright[]_{i\in I}\{b_i\&@(x_i := e_i) \to (S_i \parallel Q)\}$
par-5-5	$[]_{i\in I}\{EvtG(@(\xi_i)) \to M_i\}$ $\triangleright[]\{CbG(Cb_s) \to S\}$	$[]_{j\in J}\{EvtG(@(\eta_j)) \to N_j\}$ $\triangleright[]\{CbG(Cb_t) \to T\}$	$[]_{i\in I}\{EvtG(@(\xi_i)) \to (M_i \parallel Q)\}$ $[][]_{j\in J}\{EvtG(@(\eta_j)) \to (P \parallel N_j)\}$ $\triangleright[]\{CbG(Cb_{st}) \to ((Re(Cb_{st}, Cb_s); S)\vert\vert(Re(Cb_{st}, Cb_t); T))\}$

- **(par-1-1)** expresses that if both of the parallel components are *assignment guarded choices*, then any assignment guard can be scheduled first, and the following behavior after this selected assignment guard is the parallel composition of the subsequent process after this selected guard and another parallel part.

- **(par-1-2)** portrays the parallel composition of *assignment guarded choice* and *event guarded choice*. The schedule rule is that the assignment guard can be scheduled, and it also allows the event guard to be scheduled if this event guard is triggered. For the whole parallel process, if the assignment guard is scheduled, the subsequent behavior is the parallel composition of the following behavior of this assignment guard and another parallel part.

On the other hand, if the event guard can be triggered, the rest is the parallel composition of the process after the selected event guard and another parallel part. As introduced before, to ensure the immediate trigger of $@(\xi_j)$, $EvtG(@(\xi_j))$ is considered to have a higher priority than the assignment guards.

- **(par-1-3)** reflects the parallel composition of *assignment guarded choice* and *cb guarded choice*. Since the assignment guard is instantaneous, it can be scheduled at once. Further, cb guard is a continuous behavior that may perform for a while. For the whole parallel process, if an assignment guard is scheduled, the subsequent behavior is the parallel composition of the following process after this assignment guard and another parallel part. In general, the instantaneous assignment is considered to be executed first when we sequentialize the parallel composition of assignment and continuous behaviors.

- **(par-2-2)** stands for the situation where both parallel parts are the *event guarded choices*. Event guard is set to a higher priority than other guards, meaning that once the guard condition gd holds, the event guard can be triggered immediately. Similar to **(par-1-1)**, the parallel composition of event guarded choices is defined in an interleaving way. Any event guard can be scheduled first, and the order of triggering does not affect the final result.

- **(par-2-3)** exhibits the parallel composition of *event guarded choice* and *cb guarded choice*. Similar to **(par-1-3)**, for the whole process, if an event guard is triggered, the subsequent behavior is the parallel composition of the rest after this event guard and another parallel part. However, for the event guard, which is different from the assignment, the guard may be waiting to be triggered. Then, in this condition, the continuous behavior will perform and the subsequent behavior of the whole process is the parallel composition of the remaining process after this cb guard and another parallel part.

- **(par-3-3)** demonstrates the condition where both of the parallel parts are *cb guarded choices*. The schedule rule permits both of the cb guards (i.e., $CbG(Cb_{st})$) to perform, and the subsequent behavior is the parallel composition of the remaining continuous behaviors and their successive processes (i.e., $(Re(Cb_{st}, Cb_s); S) \parallel (Re(Cb_{st}, Cb_t); T)$).

  - Cb_{st}: Since the dynamic of a continuous variable can be defined by one and only one relation in a moment, we assume that Cb_s and Cb_t have no continuous shared variables. Cb_{st} is a conjunction of the two dynamics, i.e., the two different continuous behaviors can perform as their corresponding differential relations defined at the same time. If $Cb_s =_{df} R_1(s, \dot{s})$ **until** g_s and $Cb_t =_{df} R_2(t, \dot{t})$ **until** g_t, then $Cb_{st} =_{df} R((s,t),(\dot{s},\dot{t}))$ **until** $(g_s \vee g_t)$. Here, R merges R_1 and R_2, which indicates Cb_{st} is extended to contain multiple continuous variables.
  - $Re(Cb_{st}, Cb_s)$ and $Re(Cb_{st}, Cb_t)$: They stand for the remaining continuous behavior of Cb_s and Cb_t after executing the previous guard $CbG(Cb_{st})$. After executing $CbG(Cb_{st})$, it implies that $g_s \vee g_t$ is true. Therefore, there are three possible scenarios, and based on them, we give the definitions of $Re(Cb_{st}, Cb_s); S$ and $Re(Cb_{st}, Cb_t); T$.
    - If $g_s = true \wedge g_t = true$, then $Re(Cb_{st}, Cb_s); S = S$ and $Re(Cb_{st}, Cb_t); T = T$.
    - If $g_s = true \wedge g_t = false$, then $Re(Cb_{st}, Cb_s); S = S$ and $Re(Cb_{st}, Cb_t); T = Cb_t; T$.
    - If $g_s = false \wedge g_t = true$, then $Re(Cb_{st}, Cb_s); S = Cb_s; S$ and $Re(Cb_{st}, Cb_t); T = T$.

We take the following program as an example to explain this rule intuitively. In this example, we assume that the initial values of both s and t are 0.

$$[]\{CbG(\dot{s} = 1 \textbf{ until } s \geqslant 2) \rightarrow x := 1\} \parallel []\{CbG(\dot{t} = 2 \textbf{ until } t \geqslant 2) \rightarrow y := 1\}$$

$$= [] \left\{ \begin{array}{l} CbG((\dot{s} = 1 \wedge \dot{t} = 2) \textbf{ until } (s \geqslant 2 \vee t \geqslant 2)) \rightarrow \\ \left(\begin{array}{l} (\ Re((\dot{s} = 1 \wedge \dot{t} = 2) \textbf{ until } (s \geqslant 2 \vee t \geqslant 2), \dot{s} = 1 \textbf{ until } s \geqslant 2)\ ; x := 1) \\ || \\ (\ Re((\dot{s} = 1 \wedge \dot{t} = 2) \textbf{ until } (s \geqslant 2 \vee t \geqslant 2), \dot{t} = 2 \textbf{ until } t \geqslant 2)\ ; y := 1) \end{array} \right) \end{array} \right\}$$

$CbG((\dot{s} = 1 \wedge \dot{t} = 2)$ **until** $(s \geqslant 2 \vee t \geqslant 2))$ represents that s and t are evolving as their respective differential relations specify. After 1 time unit, s is equal to 1 and t is equal to 2. $t \geqslant 2$ is satisfied, so the continuous behavior of t terminates. Therefore, the conjunction of the dynamics of s and t also terminates. At this time, $Re((\dot{s} = 1 \wedge \dot{t} = 2)$ **until** $(s \geqslant 2 \vee t \geqslant 2), \dot{t} = 2$ **until** $t \geqslant 2); y := 1$ is equal to $y := 1$. Since the continuous behavior of s is not finished, $Re((\dot{s} = 1 \wedge \dot{t} = 2)$ **until** $(s \geqslant 2 \vee t \geqslant 2), \dot{s} = 1$ **until** $s \geqslant 2); x := 1$ is equal to $\dot{s} = 1$ **until** $s \geqslant 2; x := 1$ and the current value of s is 1.

- Algebraic laws for the parallel compositions that contain hybrid guarded choice can be derived from the above algebraic laws.

  For example, **(par-1-4)** represents the parallel composition of *assignment guarded choice* and *event&assignment hybrid guarded choice*. Based on **(par-1-1)** and **(par-1-2)**, the schedule rule is that the assignment guard from any parallel part can be scheduled. Also, the schedule rule allows the event guard to be selected if the event guard can be triggered.

  In a similar way, the algebraic laws for the remaining parallel compositions containing hybrid guarded choices can be obtained, so we omit the details here.

3.3. Usage of algebraic semantics

Theorem 1. *Every program of* CPSL^{sc} *can be transformed into the guarded choice form through the proposed algebraic semantics.*

PROOF. We prove this by structural induction. For basic statements that do not contain parallel composition, they can be converted into the guarded choice form through the algebraic laws in Section 3.2.1. For the parallel composition, by induction, the parallel components can be transformed into the guarded choice forms. Then, as enumerated in Table 2 in Section 3.2.2, the guarded choice form of the parallel composition can be derived. Therefore, every program of $CPSL^{sc}$ can be translated into the guarded choice form according to the proposed algebraic semantics. □

Based on the algebraic semantics, parallel composition of $CPSL^{sc}$ can be sequentialized. To realize the sequentialization, as mentioned above, we can get the guarded choice form of parallel composition. If the result still contains $\|$, then we can continue applying algebraic laws to obtain further parallel expansion results until all notations of $\|$ are eliminated. To better understand this theorem, an example is given to show the process of eliminating the notation $\|$ in the parallel composition. Considering the parallel composition $P \parallel Q$, where

$$P =_{df} x := 1; @(x > 1)$$
$$Q =_{df} x := 2; \dot{v} = 1 \textbf{ until } (v \geqslant 2)$$

First, we transform the parallel components P and Q into the guarded choice form. Using **(assign-1)** and **(seq-2)**, we have:

$$P = [\,]\{true\&@(x := 1) \to @(x > 1)\} \tag{1}$$
$$Q = [\,]\{true\&@(x := 2) \to \dot{v} = 1 \textbf{ until } (v \geqslant 2)\} \tag{2}$$

Since P and Q are both *assignment guarded choices*, based on **(par-1-1)**, we have:

$$P \parallel Q = [\,]\{true\&@(x := 1) \to \boxed{@(x > 1) \parallel Q}\,\} \tag{3}$$
$$[\,][\,]\{true\&@(x := 2) \to \boxed{P \parallel \dot{v} = 1 \textbf{ until } (v \geqslant 2)}\,\} \tag{4}$$

The result of $P \parallel Q$ has two branches, and they both have the notation $\|$. Thus, we continue to convert them step by step. We first focus on the parallel composition in formula (3), and apply algebraic laws to $@(x > 1) \parallel Q$. With **(guard-1)**, we have:

$$@(x > 1) = [\,]\{EvtG(@(x > 1)) \to \varepsilon\} \tag{5}$$

Because $@(x > 1)$ is an *event guarded choice* and Q is an *assignment guarded choice*, using **(par-1-2)**, **(par-0-1)** and **(par-0-3)**, we have:

$$@(x > 1) \parallel Q = [\,]\{EvtG(@(x > 1)) \to \varepsilon \parallel Q\} \tag{6}$$

$$
\begin{aligned}
&\rhd [\!]\{true\&@(x := 2) \rightarrow @(x > 1) \parallel \dot{v} = 1 \textbf{ until } (v \geqslant 2)\} && (7)\\
= {} & [\!]\{EvtG(@(x > 1)) \rightarrow Q\} && (8)\\
&\rhd [\!]\{true\&@(x := 2) \rightarrow @(x > 1) \parallel \dot{v} = 1 \textbf{ until } (v \geqslant 2)\} && (9)
\end{aligned}
$$

The result after expansion still has $\parallel$, and we keep transform $@(x > 1) \parallel \dot{v} = 1$ **until** $(v \geqslant 2)$ similarly. Using **(cb-1)**, we have:

$$
\dot{v} = 1 \textbf{ until } (v \geqslant 2) = [\!]\{CbG(\dot{v} = 1 \textbf{ until } (v \geqslant 2)) \rightarrow \varepsilon\} \qquad (10)
$$

Then, applying **(par-2-3)** and **(par-0-3)**, we have:

$$
\begin{aligned}
@(x > 1) \parallel \dot{v} = 1 \textbf{ until } (v \geqslant 2) = {} & [\!]\{EvtG(@(x > 1)) \rightarrow \varepsilon \parallel \dot{v} = 1 \textbf{ until } (v \geqslant 2)\} && (11)\\
&\rhd [\!]\{CbG(\dot{v} = 1 \textbf{ until } (v \geqslant 2)) \rightarrow @(x > 1) \parallel \varepsilon\} && (12)\\
= {} & [\!]\{EvtG(@(x > 1)) \rightarrow \dot{v} = 1 \textbf{ until } (v \geqslant 2)\} && (13)\\
&\rhd [\!]\{CbG(\dot{v} = 1 \textbf{ until } (v \geqslant 2)) \rightarrow @(x > 1)\} && (14)
\end{aligned}
$$

Therefore, according to the proposed algebraic laws, the formula (3) in $P \parallel Q$ is equivalent to the following guarded choice form.

$$
(3) = [\!]\left\{\begin{array}{l}
true\&@(x := 1) \rightarrow \\
\qquad \left(\begin{array}{l}
[\!]\{EvtG(@(x > 1)) \rightarrow Q\} \\
\rhd [\!]\left\{ true\&@(x := 2) \rightarrow \left(\begin{array}{l}
[\!]\{EvtG(@(x > 1)) \rightarrow \dot{v} = 1 \textbf{ until } (v \geqslant 2)\} \\
\rhd [\!]\{CbG(\dot{v} = 1 \textbf{ until } (v \geqslant 2)) \rightarrow @(x > 1)\}
\end{array}\right) \right\}
\end{array}\right)
\end{array}\right\}
$$

Likewise, the formula (4) in $P \parallel Q$ can be expanded, and here we omit the details. Then, we can obtain the final expansion form of $P \parallel Q$ without $\parallel$, and the sequentialization is realized.

4. Mechanizing Algebraic Laws in Maude

In this section, we apply the rewriting engine Real-Time Maude to implement the algebraic semantics. We first define the programs, guarded components and guarded choices of $CPSL^{sc}$ in Section 4.1. Then, in Section 4.2, we introduce an operator called `GCF` to derive the guarded choice form from the program. Moreover, considering the specific execution depends on the specific data state, we adopt an operator called `EXE` to realize the execution of the program in Section 4.3. Finally, to bring out the automatic simulation and verification, operators named `autoTransExe` and `trVerify` are defined in Section 4.4 and Section 4.5.

4.1. Implementation of programs and guarded choices

In this subsection, we formalize the programs and guarded choices of $CPSL^{sc}$ using the Real-Time Maude. For preparation, we first give the basic sorts of our formalization, including variable `Var`, expression `Exp`, Boolean expression `Boolexp`, guard condition `Guard` and program `Program`. Besides, to mechanize the guarded choices, we define sorts of algebraic guards `algGuard`, guarded components `guardedComp` and guarded choices `guardedChoice`.

More specifically, `DVar` and `CVar` stand for discrete variables and continuous variables respectively, and they are subsorts of `Var`. `Exp` represents an expression that is evaluated and assigned to discrete variables in the assignment. `Boolexp` appears in the conditional construct and iteration construct. For guard condition `Guard`, it has two subsorts (i.e., `Gd` and `Gc`). Here, we only list the declaration and the relation of these sorts.

```
sorts Var Exp Boolexp Guard Program .
sorts algGuard guardedComp guardedChoice .

---Variable
sorts DVar CVar .
subsorts DVar CVar < Var .

---Guard Condition
sorts Gd Gc .
subsorts Gd Gc < Guard .
```

4.1.1. *Program*

After introducing the above basic definitions, we implement the program of *CPSL^sc^* as below. We define three sorts as the basic statements (i.e., `Assignment`, `Event` and `Cb`), and they are subsorts of `Program`. It is worthy of noting that we use the form of `R(v '= r) until g` to describe the continuous behaviors in Maude. Here, `v` is the continuous variable described currently, `r` stands for the rate of evolution, and `g` is the guard condition.

We formalize these basic statements and composition of them through operators `op` in Maude, and we set different precedence values and gathering patterns for them. For example, the definition in Line 13 indicates that the parallel composition of two programs is still a program, and it satisfies the commutative and associative attributes.

```
1  sorts Assignment Event Cb .
2  subsorts Assignment Event Cb < Program .
3
4  ---Basic Statement
5  op _:=_ : DVar Exp -> Assignment [ctor prec 1] .
6  op @`(_`) : Gd -> Event [ctor prec 1] .
7  op R`(_'=_`)until_ : CVar Rat Guard -> Cb [ctor prec 1] .
8
9  ---Composition
10 op _;_ : Program Program -> Program [ctor assoc prec 2] .
11 op if_then_else_ : Boolexp Program Program -> Program [ctor prec 2] .
12 op while_do_ : Boolexp Program -> Program [ctor prec 2] .
13 op _||_ : Program Program -> Program [ctor assoc comm prec 2] .
```

4.1.2. *Guarded choice*

As explained before, $h \to P$ is a guarded component if h is an algebraic guard. Therefore, we first construct three types of algebraic guards (i.e., `assignGuard`, `gdGuard` and `cbGuard`). Based on them, we implement three guarded components of *CPSL^sc^*, including `assignGComp`, `eventGComp` and `cbGComp`. Next, we give the mechanical definitions of guarded choices in Maude as below. It is worth noting that, for the sake of convenience in mechanization, we consistently use [] instead of $\triangleright$ defined in Section 3.1.3. To ensure the priority of event guards, the corresponding rule is defined in Section 4.3.4.

```
---Algebraic Guard
sorts assignGuard eventGuard cbGuard .
subsorts assignGuard eventGuard cbGuard < algGuard .
op _&@_ : Bool Assignment -> assignGuard [ctor] .
op EvtG`(_`) : Event -> eventGuard [ctor] .
op CbG`(_`) : Cb -> cbGuard [ctor] .

---Guarded Component
sorts assignGComp eventGComp cbGComp .
subsorts assignGComp eventGComp cbGComp < guardedComp .
op _->_ : assignGuard Program -> assignGComp [ctor] .
op _->_ : eventGuard Program -> eventGComp [ctor] .
op _->_ : cbGuard Program -> cbGComp [ctor] .

---Guarded Choice
sorts assignGC eventGC cbGC .
subsorts assignGC eventGC cbGC < guardedChoice .
op `[`]`{_`} : assignGComp -> assignGC [ctor] .
op `[`]`{_`} : eventGComp -> eventGC [ctor] .
op `[`]`{_`} : cbGComp -> cbGC [ctor] .
op _`[`]_ : guardedChoice guardedChoice -> guardedChoice [ctor comm assoc] .
```

4.2. Derivation of guarded choice form from programs

In this subsection, we adopt the operator `GCF` to transform *CPSL^sc^* into the guarded choice form.

4.2.1. Basic statements

According to the proposed algebraic laws, we first study the laws for basic statements. Here, E stands for an empty program. `asg`, `evt` and `cbeh` denote the variables of the sort `Assignment`, `Event` and `Cb` respectively. `checkb` is used to check whether the Boolean expression `bexp` is true or not at a corresponding data state, and ∼`checkb` is the negation of `checkb`.

```
op GCF`(_`) : Program -> guardedChoice [ctor] .

---Baisc Statement
eq GCF(asg) = []{true &@(asg) -> E} .
eq GCF(evt) = []{EvtG(evt) -> E} .
eq GCF(cbeh) = []{CbG(cbeh) -> E} .

---Sequential Composition
eq GCF(P ; Q) = GCF(P) ; Q .
eq []{alg -> P} ; Q = []{alg -> (P ; Q)} .
eq ([]{alg -> P} [] gch) ; Q = ([]{alg -> P} ; Q)[](gch ; Q) .

---Conditional and Iteration Constructs
eq GCF(if bexp then P else Q) = []{checkb(bexp)&@(skip)-> P}
                                []{(~ checkb(bexp))&@(skip)-> Q} .
eq GCF(while bexp do P) = []{checkb(bexp)&@(skip)-> (P ; while bexp do P)}
                          [][]{(~ checkb(bexp))&@(skip)-> E} .
```

4.2.2. Parallel composition

Then, we look into algebraic laws of the parallel composition for guarded choices. As analyzed previously, we formalized 15 expansion laws in Maude. Here, we only list some of them as examples, the remaining mechanization can be found in Appendix A.1. These mechanized definitions are consistent with their theoretical definitions. We apply the pattern match as a condition in these equations to determine what types of guarded choices of the parallel components (i.e., P and Q) are, and then decide the guarded choice form of the parallel composition of P and Q. More specifically, the key symbol $:=$ compares a pattern on the left-hand side with the right-hand side, and if they match, returns true.

```
1 ---(par-1-1)
2 ceq GCF(P || Q) = par1(asggc1,Q)[]par1(asggc2,P)
3                 if asggc1 := GCF(P) /\ asggc2 := GCF(Q) .
4
5 ---par1: One is Assginment Guarded Choice
6 op par1`(_`,_`) : assignGC Program -> assignGC .
7 eq par1([]{b &@ (asg)-> P'},Q) = []{b &@ (asg)->(P' || Q)} .
8 eq par1([]{asgg1 -> P'}[]asggc2,Q) = par1([]{asgg1 -> P'},Q)[]par1(asggc2,Q) .
```

We take **(par-1-1)** as an example. As shown in Line 2, it implies that if P and Q are both assignment guarded choices, then their parallel composition is also an assignment guarded choice. In the above formalization, we apply the operators $par*$ to compute the components of parallel expansions of two processes. The process which has not been scheduled will be added as a parameter to the computation, so that the remaining process after the corresponding guard of the parallel expansion can be computed. The complete definition of $par*$ is presented in Appendix A.2. Here, $par1$ is used to compute the components of parallel composition where one process is an assignment guarded choice.

Next, we present the formalization of **(par-3-3)**. As mentioned in Section 3.2.2, both of the cb guarded choices have the chance to perform. In Maude, we use $par33$ to explain the case where both parallel components are cb guarded choices. Here, `Merge` stands for the conjunction of the two dynamics and `Re` denotes the remaining continuous behavior after the previous conjunction. The mechanization is consistent with the previous definition in Section 3.2.2.

```
---(par-3-3)
ceq GCF(P || Q) = par33(cbgc1,cbgc2) if cbgc1 := GCF(P) /\ cbgc2 := GCF(Q) .

```

```
---par33: Both are Cb Guarded Choices
op par33`(_`,_`) : cbGC cbGC -> cbGC .
eq par33([]{cbg1 ->S},[]{cbg2 -> T}) =[]{Merge(cbg1 , cbg2)->
   ((Re(Merge(cbg1,cbg2),cbg1); S)||(Re(Merge(cbg1,cbg2),cbg2); T))} .

---Definitionn of Merge and Re
op Merge`(_`,_`) : cbGuard cbGuard -> cbGuard [ctor comm] .
op Re`(_`,_`) : cbGuard cbGuard -> Program [ctor] .
eq Merge(CbG(R(s '= m) until g1),CbG(R(t '= n) until g2))
  = CbG(R((s , t) '= (m , n)) until (g1 \/ g2)) .
```

4.3. Execution of programs with the guarded choice form

In this subsection, after gaining the guarded choice form from the program, we now explore how to execute programs with the guarded choice form in Maude.

Considering that the specific execution of the program depends on the specific data state, to record the data state, we first introduce a sort `Statev`. `Statev` is defined to record one variable's data state. We assume there are five variables in this paper, and construct a tuple like `[x <- m,y <- n,z <- g,u <- k,v <-l]` to record their values. This form of tuple constitutes a trace (defined as the sort `Trace`), and concatenation can be realized through the operator ˆ. We define the sort `State` which is the subsort of the whole system `System` to represent the current execution state. We construct a `Trace` combined with a `Program` as a `State`. The current data state is stored in `Trace`, while `Program` represents the programs that have not been executed yet. Additionally, when the program finishes execution, there are no programs to be executed later. Hence, we define `Trace` as the subsort of `State`.

Then, we introduce an operator `EXE` to realize the transformation from the program with the guarded choice form to the data state, i.e., executing the program in one step. `EXE(pre:GCF(P))` denotes the execution of the program P under the current trace `pre`. Note that the program P here is the guarded choice form. `pre` is a trace which contains a sequence of data states, and the last tuple of `pre` stands for the current data state.

In a nutshell, algebraic semantics is used to decide execution order (getting the first action), especially in realizing the sequentialization of parallel composition through the transformed guarded choice form. Then, based on guarded choice form, the transition of these first actions follow the operational semantics proposed in [7] which is appended in Appendix B.

```
sorts Statev Trace State .
subsort Trace < State .
subsort State < System .
op _:_ : Trace Program -> State .

---Definition of Trace
op _<-_ : Var Rat -> Statev [ctor] .
op `[_`,_`,_`,_`,_`] : Statev Statev Statev Statev Statev -> Trace [ctor] .
op _^_ : Trace Trace -> Trace [ctor assoc] .

op EXE`(_:_`) : Trace guardedChoice -> State [ctor] .
```

4.3.1. Assignment guarded choice

For the assignment guarded choice, only when the Boolean condition b of the guarded choice is `true`, the assignment can be executed. Here, `last(pre)` is defined to get the last snapshot of the trace `pre`. The operator `assign` is defined to calculate and record the result of the assignment. Thus, after executing the assignment guarded choice, the result will be added to the tail of `pre`. Otherwise, the program terminates at the current trace `pre`.

```
crl EXE(pre :[]{b &@ (asg)-> P}) => (pre ^ assign(last(pre) : asg)): P
                                   if b == true .
crl EXE(pre :[]{b &@ (asg)-> P}) => pre if b == false .
```

4.3.2. Event guarded choice

For the event guarded choice, once the previous state can trigger the guard (i.e., `check(gd1,last(pre)) == true`), this guarded choice is executed. Then, the trace `pre` keeps unchanged, and the program tends to `P`. Otherwise, the program waits for the environment to trigger the guard. `wait` is declared as a special state, representing the event guard cannot be triggered at present and it waits for the environment to trigger.

```
crl EXE(pre :[]{EvtG(@(gd1))-> P}) => pre : P if check(gd1,last(pre)) == true .
crl EXE(pre :[]{EvtG(@(gd1))-> P}) => pre ^ wait
                                       if check(gd1,last(pre)) == false .
```

4.3.3. Cb guarded choice

For the continuous behavior guarded choice, the operator `evolve` is used to implement the procedure of evolution in Maude. Before evolving, as presented in Line 2, we first obtain the initial value of the continuous variable `v` through `getv(pre:v)`. After getting the initial value, the continuous behavior starts to evolve as the differential relation specifies. The tick rewrite rule in Line 4 describes the evolution process. `checkR(pre,t0,gv)` is applied to check whether the current data state can trigger `gv`. If `gv` has not been triggered (i.e., `checkR(pre,t0:gv) == false`), `v` changes by the rate `r` per time unit. Here, as mentioned in Section 2, `td` stands for the duration, `t0` and `t0+(r*td)` indicates the initial and terminal value of `v`. Once `gv` is triggered (i.e., `checkR(pre,t0:gv) == true`), the value of the continuous variable is recorded through the operator `record`.

```
1  ---Evolve
2  rl EXE(pre :[]{CbG(R(v '= r) until gv)-> P}) =>
3     (pre ^ evolve(R(v '= r) until gv)[getv(pre : v)]) : P .
4  crl {(pre ^ evolve(R(v '= r) until gv)[t0]) : P} =>
5      {(pre ^ evolve(R(v '= r) until gv)[t0 + (r * td)]) : P} in time td
6      if checkR(pre,t0 : gv) == false [nonexec] .
7
8  ---Record Values
9  ceq pre ^ evolve(R(v '= r) until gv)[t0]  =
10     pre ^ (record(last(pre) : evolve(R(v '= r) until gv)[t0]))
11     if checkR(pre,t0 : gv)  == true .
```

Based on the above formalization, then we mechanize the parallel composition of two continuous behaviors. As listed in **(par-3-3)** of Table 3, we need to describe the evolution of the conjunction of two dynamics. We first implement the initiation of evolution (i.e., get the initial value of continuous variables) as listed in Line 2-4.

Then, we adopt the tick rewrite rules to define the evolution process. Line 5-8 describes that both continuous behaviors can perform as their differential relation specifies, i.e., u and v increase by the rate $r1$ and $r2$ per time unit respectively.

After evolving, once one of the continuous behaviors terminates, we record the terminal values of the continuous variables using the operator `record`.

Finally, the mechanical implementation of Re in Maude is given. $Re(CbG(cb_{uv}), CbG(cb_u))$ stands for the remaining continuous behavior of cb_u after the conjunction of cb_{uv}. According to the termination order of cb_u and cb_v, we give the equations of Re. The first one denotes that both continuous behaviors terminate at the same time. The second one implies that the continuous behavior of u terminates first, while the last one means the continuous behavior of v terminates earlier than the behavior regarding u.

```
1  ---Evolution of the conjunction of two dynamics
2  rl EXE(pre :[]{CbG(R(u,v '=  r1,r2) until (gu \/ gv))-> P}) =>
3            (pre ^ evolve(R(u,v '= r1,r2) until (gu \/ gv))
4            [getv(last(pre): u),getv(last(pre): v)]) : P .
5  crl {(pre ^ evolve(R(u,v '= r1,r2) until (gu \/ gv))[t0,t1]) : P} =>
6      {(pre ^ evolve(R(u,v '= r1,r2) until (gu \/ gv))
7      [t0 + (r1 * td) , t1 + (r2 * td)]) : P} in time td
8    if checkR(pre,t0 : gu) == false /\ checkR(pre, t1 : gv) == false [nonexec] .
9
10 ---Record values
```

```
ceq pre ^ evolve(R(u,v '= r1,r2) until (gu \/ gv))[t0,t1] = pre ^
   (record(last(pre) : evolve(R(u,v '= r1,r2) until (gu \/ gv))[t0,t1]))
   if (checkR(pre,t0 : gu ) == true) \/ (checkR(pre,t1 : gv ) == true) .

----Definition of Re
----R(u) and R(v) both terminate
ceq EXE(pre : GCF((Re(CbG(R(u,v '= r1,r2)until(gu \/ gv)),CbG(R(u '= r1)until
    gu)); P)||(Re(CbG(R(u,v '= r1,r2)until(gu \/ gv)),CbG(R(v '= r2)until gv));
     Q))) = EXE(pre : GCF(P || Q))
     if checkR(pre,getv(pre : u) : gu) == true /\
        checkR(pre,getv(pre : v) : gv) ==  true .
----R(v) continues and R(u) terminates
ceq EXE(pre : GCF((Re(CbG(R(u,v '= r1,r2)until(gu \/ gv)),CbG(R(u '= r1)until
    gu)); P)||(Re(CbG(R(u,v '= r1,r2)until(gu \/ gv)),CbG(R(v '= r2)until gv));
     Q))) = EXE(pre : GCF(P ||(R(v '= r2 )until gv ; Q)))
     if checkR(pre,getv(pre : u) : gu) == true /\
        checkR(pre,getv(pre : v) : gv) ==  false  .
----R(u) continues and R(v) terminates
ceq EXE(pre : GCF((Re(CbG(R(u,v '= r1,r2)until(gu \/ gv)),CbG(R(u '= r1)until
    gu)); P)||(Re(CbG(R(u,v '= r1,r2)until(gu \/ gv)),CbG(R(v '= r2)until gv));
     Q))) = EXE(pre : GCF((R(u '= r1)until gu ; P)|| Q))
     if checkR(pre,getv(pre : u) : gu) == false /\
        checkR(pre,getv(pre : v) : gv) == true  .
```

4.3.4. *Auxiliary rules and equations*

After giving the execution for the above commands, we can easily execute the compositional commands through the algebraic laws. However, to complete the execution of $CPSL^{sc}$, we also need to add the following rules. They can transform `checkb(bexp)` to `ckbexp(bexp, last(pre))` in conditional construct and iteration construct, so that whether the Boolean expression *bexp* is satisfied under the current data state `last(pre)` can be determined.

```
rl EXE(pre : []{checkb(bexp) &@ (skip) -> P}) =>
   EXE(pre : []{ckbexp(bexp,last(pre)) &@ (skip) -> P}) .
rl EXE(pre : []{~ checkb(bexp) &@ (skip) -> P}) =>
   EXE(pre : []{(not ckbexp(bexp,last(pre))) &@ (skip) -> P}) .
```

Additionally, considering that there are several execution orders under the parallel composition, we add the operator $\vee$ to formalize these situations where the whole program can execute the guarded choice either $gch1$ or $gch2$.

```
op _\/_ : State State -> State [ctor comm  assoc] .
rl EXE(pre : gch1[]gch2) => EXE(pre : gch1) \/ EXE(pre : gch2) .
```

In Section 3.1.3, we set event guarded choice as an urgent guarded choice. In Maude, to ensure the priority of scheduling, a rewrite rule is defined where `nualg` stands for `assignGuard` and `cbGuard`.

```
crl EXE(pre : []{EvtG(@(gd1))-> P}) \/ EXE(pre : []{nualg -> Q})
        => pre : P if check(gd1,last(pre)) == true .
```

The equations below are used for reduction. If one branch of the execution result is `wait` while another can continue to execute, then the branch of `wait` is reduced.

```
op wait : -> State [ctor] .
eq (pre ^ wait) \/ (pre ^ pre') = pre ^ pre' .
eq (pre ^ pre' ^ wait) \/ (pre ^ wait) = pre ^ pre' ^ wait .
```

To get a better understanding of the previous formalization of `EXE(pre :[]EvtG(@(gd1))-> P)` and the reduction for `wait`, we consider the program $x := 2 \parallel @(x > 1)$ and the initial state is `[x<-0]`. By applying the rewrite commands `GCF` and `EXE`, we can obtain the final execution result. As illustrated in Figure 2, the detailed steps are as follows.

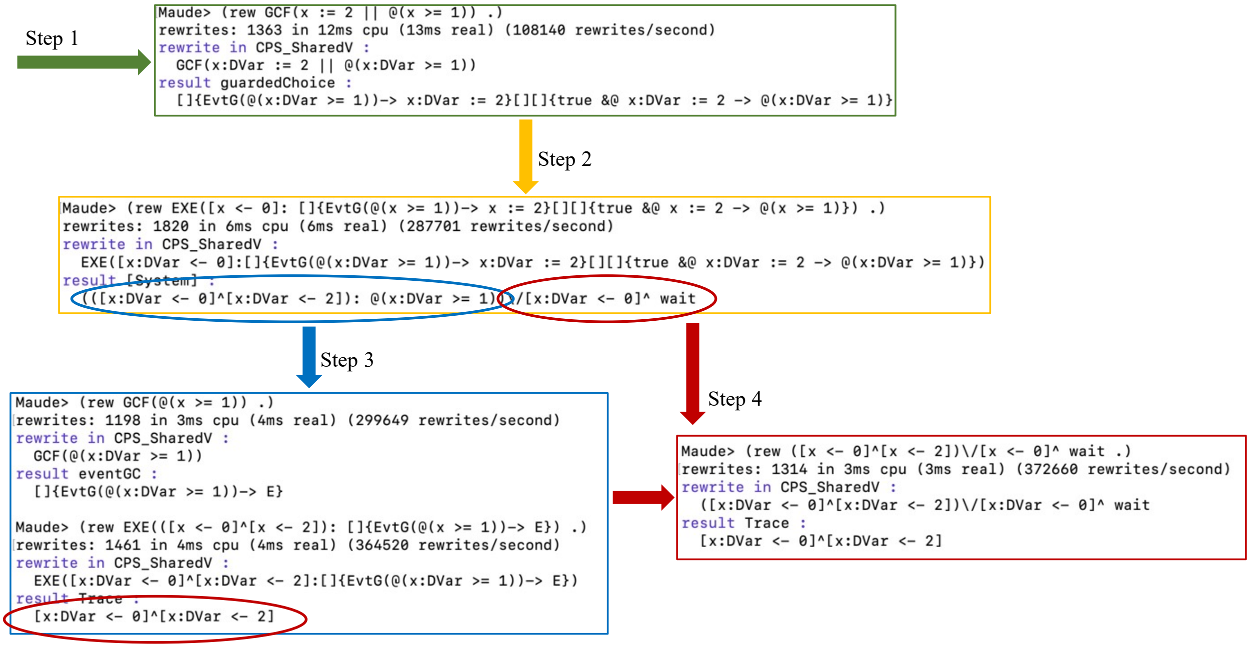

Figure 2: Execution of $x := 2 \parallel @(x > 1)$

- **Step 1:** Through rewriting `GCF`, the guarded choice form of $x := 2 \parallel @(x > 1)$ is obtained. The result shows it is an event&assignment hybrid guarded choice.
- **Step 2:** Combining with the initial state `[x<-0]` and applying the operator `EXE`, the program executes one step. The result consists of two branches, which represent the execution of assignment guarded choice and event guarded choice respectively. The former has $@(x > 1)$ to be executed subsequently, while the latter enters the `wait` state.
- **Step 3:** We continue to employ the rewriting commands `GCF` and `EXE` for the previous branch and obtain the execution result `[x<-0]^[x<-2]`.
- **Step 4:** Finally, we can get the final execution result due to the reduction equation.

4.4. Automatic transformation and execution

Although the program can be simulated by the above commands step by step, it is obvious that this manual approach is laborious. Therefore, in this subsection, we introduce an operator `autoTransExe` to automate the transformation from the program to its guarded choice form and then realize the automatic execution. Here, `pre - last(pre)` stands for the rest after deleting the last data state of `pre`. Due to the automatic transformation and execution, for a given program and its initial data state, Maude can return a trace of data states generated during the execution of the program. As a result, the automatic simulation is achieved.

```
op autoTransExe '(_:_') : Trace Program -> State .
rl autoTransExe(pre : P) => EXE(pre : GCF(P)) .
crl pre : P => (pre - last(pre)) ^ autoTransExe(last(pre) : P)
    if [x <- m,y <- n,z <- g,u <- k,v <- l] := last(pre) .
```

4.5. Verification from a trace view

In essence, the core of transformation from the program to the guarded choice form is to obtain the first action of the given program. As a consequence, the sequentialization of parallel programs is accomplished. Then, with the adhered initial data state, the program can execute step by step and generate the corresponding trace which comprises data states during its runtime.

This idea coincides with the nature of runtime verification which allows checking whether a run of a system meets or violates a given property [13]. In the runtime verification, the checking is typically performed using a *monitor* which reads a finite trace and yields a certain verdict. Inspired by this, we follow the concept of online monitor [13] and carry out our verification through a similar but invisible monitor which is responsible for checking the current execution of a system. Before verifying programs, we first declare a new sort `Prop` representing the property to be verified. In this paper, as defined in the operator `Exist`, only the reachability property is described.

```
sort Prop .
op Exist`(_`) : Boolexp -> Prop [ctor] .
op verify`(_`,_`) : Prop State -> Bool [ctor] .
ceq verify(Exist(bexp),pre) = true if ckbexp(bexp , pre) == true .
ceq verify(Exist(bexp),pre) = false if ckbexp(bexp , pre) == false .
```

Now, similar to the previous codes of `autoTransExe`, we present the operator `trVerify` with its rewrite rules and equations to conduct verification. `FirstExistence` and `NoExistence` are two special states. `FirstExistence` denotes the data state that first causes the property `sp` to be satisfied, and `NoExistence` implies no data state meets `sp`. As formalized in Line 3-6, if the current state cannot meet `sp`, the program continues to execute and the invisible monitor continues to check whether the newly added data state can satisfy `sp`. Once `verify(sp,last(pre))` equals to `true`, the procedure of verification terminates and returns the current data state following with a notation `FirstExistence`. Otherwise, when the program terminates and none of the data states during its runtime meets `sp`, we set the final result as `NoExistence`.

```
1  ---Definition of trVerify
2  op trVerify`(_^_`) : State Prop -> State [ctor] .
3  crl trVerify((pre : P) ^ sp) => EXE(pre : GCF(P)) ^ sp
4          if verify(sp,last(pre)) == false .
5  crl (pre : P) ^ sp => trVerify((last(pre) : P) ^ sp)
6          if  [x <- m,y <- n,z <- g,u <- k,v <- l] := last(pre) .
7
8  ---Yield verdict
9  ops FirstExistence NoExistence : -> State [ctor] .
10 ceq pre ^ sp = NoExistence if verify(sp,last(pre)) == false .
11 ceq trVerify((pre : P)^ sp) = pre ^ FirstExistence
12         if verify(sp,last(pre)) == true .
```

5. Case Study

In this section, we present a case study of Battery Management System (BMS) to showcase the simulation and verification in Real-Time Maude. BMS is abstracted from the real-world problem and it is easy to understand, in this paper, BMS is chosen as an example to illustrate the applicability of our proposed approach.

We first give the BMS program to demonstrate the syntax of $CPSL^{sc}$. Then, we enter the program in the terminal of Maude and analyze the results obtained from simulation and verification. Finally, a discussion about the approach and limitation is provided.

5.1. Overview of BMS

BMS is an important component of the electric vehicle power battery system. We employ the process of heat management in BMS as a case study. For simplicity, we assume that the battery works properly if its temperature is between 80 and 90. When the vehicle is moving and the temperature does not exceed 100, the temperature increases linearly at the rate of 1. The controller of BMS will cool down the battery if the temperature is equal to or greater than 90. Also, the temperature decreases linearly at the rate of 2 when the controller is cooling and the temperature does not reach 0. If the temperature is equal to or less than 80, the controller will stop cooling. The detailed program of BMS is presented below.

$BMS =_{df} Ctrl \parallel Temp;$

$\quad Ctrl =_{df}$ **while** $DT < 10$ **do**

$$\left\{\begin{array}{l}\dot{u}=1\ \textbf{until}\ u \geqslant DT+1;\\ \textbf{if}\ (v \geqslant 90)\ \textbf{then}\ coolon := 1;\ \textbf{else}\ skip;\\ \textbf{if}\ (v \leqslant 80)\ \textbf{then}\ coolon := 0;\ \textbf{else}\ skip;\\ DT := u;\end{array}\right\}$$

$$caron := 0;$$

$$Temp =_{df} \textbf{while}\ DT < 10\ \textbf{do}$$

$$\left\{\begin{array}{l}@(caron=1);\\ \textbf{if}\ (coolon == 0)\ \textbf{then}\ \dot{v}=1\ \textbf{until}\ ((coolon=1 \vee caron=0) \vee v \geqslant 100);\\ \qquad\qquad\qquad\quad \textbf{else}\ \dot{v}=-2\ \textbf{until}\ ((coolon=0 \vee caron=0) \vee v \leqslant 0);\end{array}\right\}$$

There are two main parts of *BMS*. *Ctrl* is the process that monitors and controls the temperature through the discrete variables `caron` and `coolon`. `Temp` describes the evolution of temperature, and affects the process `Ctrl` through the continuous variable v. Here, `caron` represents the switch of the car. When `caron=0`, we assume all behaviors terminate. `coolon` stands for the switch of cooling, BMS starts to cool down the battery if `coolon=1`. The continuous variable v records the value of the temperature. The continuous variable u and the discrete variables DT are auxiliary variables defined to record time.

5.2. Simulation and Verification

In this subsection, we simulate and verify the program of BMS through the rewrite commands in Maude. Based on the formalization in Section 4, we set the initial data state and the program BMS below.

- Initial state (initS): `[DT <- 0,caron <- 1,coolon <- 0,u <- 0,v <- 89]`
- Program (BMSP): `(while DT < 10 do(R(u '= 1) until (u >= (DT + 1)) ; if v >= 90 then coolon := 1 else skip ; if v <= 80 then coolon := 0 else skip ; DT := u); caron := 0) || (while DT < 10 do (@(caron = 1); if coolon == 0 then R(v '= 1)until((coolon = 1 \/ caron = 0) v >= 100)else R(v '= -2)until((coolon = 0 \/ caron = 0) \/ v <= 0)))`

5.2.1. Simulation

To demonstrate the decomposition and reduction of the parallel program BMS, we can get the guarded choice form of BMS with the following command, and the parameter here is BMSP we defined above.

```
(rew GCF(BMSP) .)
```

The result is obtained below.

$[]\{checkb(DT < 10)\&@skip \rightarrow Ctrl \parallel (P_t;Temp)\}[][]\{\sim checkb(DT < 10)\&@skip \rightarrow Ctrl\}$
$[][]\{checkb(DT < 10)\&@skip \rightarrow (P_c;Ctrl) \parallel Temp\}[][]\{\sim checkb(DT < 10)\&@skip \rightarrow caron := 0 \parallel Temp\}$,
Here, P_t and P_c stand for the loop body of $Temp$ and $Ctrl$. The detailed result is shown in Figure 3. For the parallel program BMS, the beginning of both the parallel components $Temp$ and $Ctrl$ is a loop. According to the algebraic law **(iter-1)**, they can be transformed into the form of assignment guarded choices. Then, the guarded choice form of `BMS` is also an assignment guarded choice and it conforms with the algebraic law **(par-1-1)**.

Further, using the timed rewrite command below, we can execute the guard choice (i.e., the program executes one step).

```
(trew EXE(initS : GCF(BMS)) with no time limit .)
```

As presented in Figure 4, The first half of the result indicates that `Temp` executes "at first", while the rest implies that `Ctrl` performs "before" `Temp` when the parallel composition is sequentialized. Here, although the parallel components actually execute simultaneously, "at first" and "before" stand for the order we recorded in the trace when we sequentialize the parallel programs. Following these steps, simulation of the execution of the program is gained manually.

Then, we feed the parallel program combined with its initial data state in Maude. By means of the following command, Maude returns a trace of the data state.

```
(trew autoTransExe(initS : BMSP) with no time limit .)
```

As shown in Figure 5, The trace is generated during the execution, and we can find that the evolution of variables in BMS meets our expectations, i.e., automatic simulation is achieved.

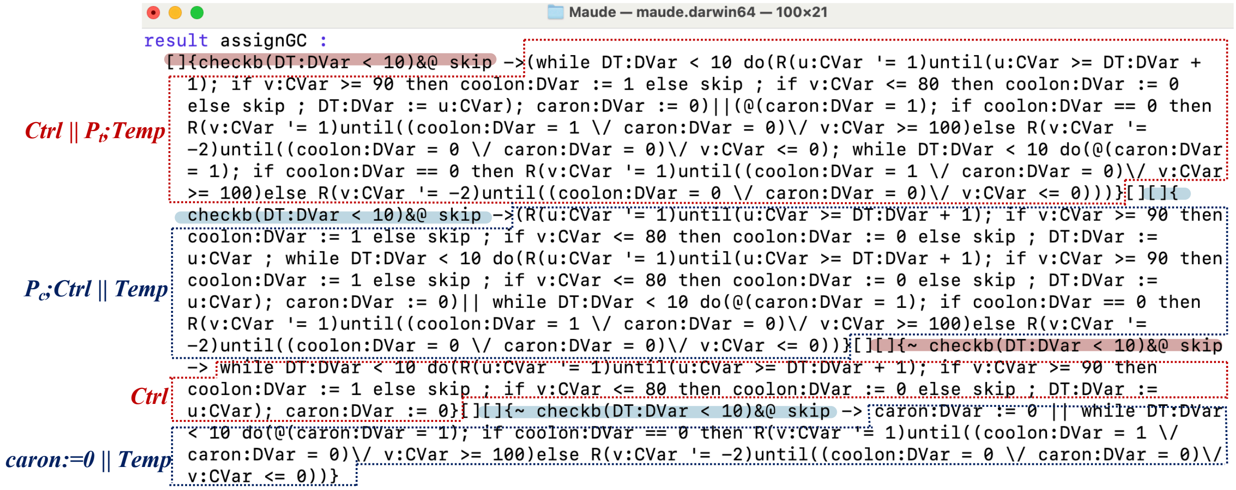

Figure 3: Getting guarded choice form in Maude

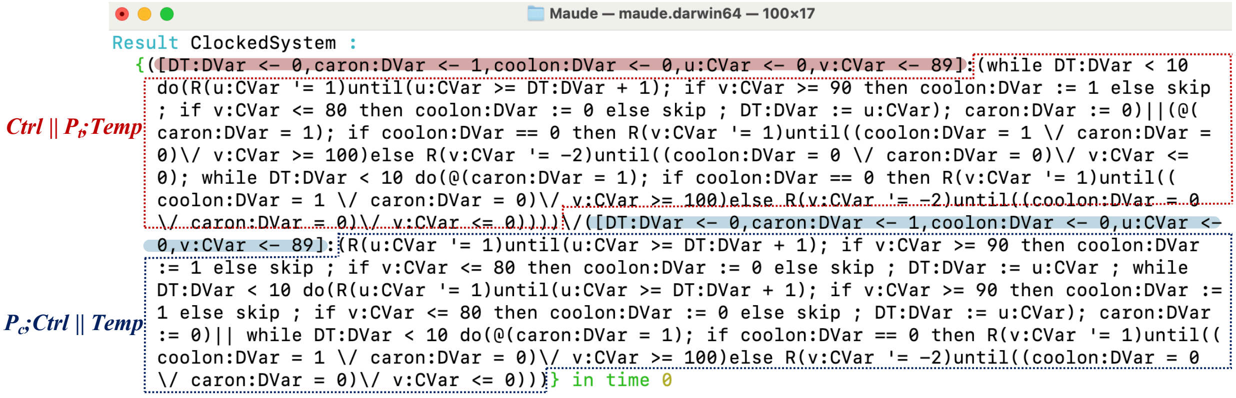

Figure 4: Executing the guarded choice form in Maude

5.2.2. *Verification*

We verify whether the temperature of the battery is guaranteed to be controlled within safe limits (i.e., v is never lower than 80 and higher than 90) through the following two commands.

```
(trew trVerify(initS : BMSP ^ Exist(v < 80)) with no time limit .)
(trew trVerify(initS : BMSP ^ Exist(v > 90)) with no time limit .)
```

The returned results shown in Figure 6(a) and Figure 6(b), we can draw a conclusion that the temperature cannot exceed the safe range when the controller of BMS works.

Moreover, we also check whether the system can reach the state where $v \leqslant 88$ with the following command.

```
(trew trVerify(initS : BMSP ^ Exist(v <= 88)) with no time limit .)
```

As presented in Figure 6(c), the returned result indicates that v will drop to 88 during the run of the system. The returned result contains a data state, which is the state when v is less than or equal to 88 for the first time. It can be found that the result returned by the verification is consistent with the simulation result in Figure 5.

```
Maude — maude.darwin64 — 81×26
Result ClockedSystem :
  {[DT:DVar <- 0,caron:DVar <- 1,coolon:DVar <- 0,u:CVar <- 0,v:CVar <- 89]^[
    DT:DVar <- 0,caron:DVar <- 1,coolon:DVar <- 0,u:CVar <- 1,v:CVar <- 90]^[
    DT:DVar <- 0,caron:DVar <- 1,coolon:DVar <- 1,u:CVar <- 1,v:CVar <- 90]^[
    DT:DVar <- 1,caron:DVar <- 1,coolon:DVar <- 1,u:CVar <- 1,v:CVar <- 90]^[
    DT:DVar <- 1,caron:DVar <- 1,coolon:DVar <- 1,u:CVar <- 2,v:CVar <- 88]^[
    DT:DVar <- 2,caron:DVar <- 1,coolon:DVar <- 1,u:CVar <- 2,v:CVar <- 88]^[
    DT:DVar <- 2,caron:DVar <- 1,coolon:DVar <- 1,u:CVar <- 3,v:CVar <- 86]^[
    DT:DVar <- 3,caron:DVar <- 1,coolon:DVar <- 1,u:CVar <- 3,v:CVar <- 86]^[
    DT:DVar <- 3,caron:DVar <- 1,coolon:DVar <- 1,u:CVar <- 4,v:CVar <- 84]^[
    DT:DVar <- 4,caron:DVar <- 1,coolon:DVar <- 1,u:CVar <- 4,v:CVar <- 84]^[
    DT:DVar <- 4,caron:DVar <- 1,coolon:DVar <- 1,u:CVar <- 5,v:CVar <- 82]^[
    DT:DVar <- 5,caron:DVar <- 1,coolon:DVar <- 1,u:CVar <- 5,v:CVar <- 82]^[
    DT:DVar <- 5,caron:DVar <- 1,coolon:DVar <- 1,u:CVar <- 6,v:CVar <- 80]^[
    DT:DVar <- 5,caron:DVar <- 1,coolon:DVar <- 0,u:CVar <- 6,v:CVar <- 80]^[
    DT:DVar <- 6,caron:DVar <- 1,coolon:DVar <- 0,u:CVar <- 6,v:CVar <- 80]^[
    DT:DVar <- 6,caron:DVar <- 1,coolon:DVar <- 0,u:CVar <- 7,v:CVar <- 81]^[
    DT:DVar <- 7,caron:DVar <- 1,coolon:DVar <- 0,u:CVar <- 7,v:CVar <- 81]^[
    DT:DVar <- 7,caron:DVar <- 1,coolon:DVar <- 0,u:CVar <- 8,v:CVar <- 82]^[
    DT:DVar <- 8,caron:DVar <- 1,coolon:DVar <- 0,u:CVar <- 8,v:CVar <- 82]^[
    DT:DVar <- 8,caron:DVar <- 1,coolon:DVar <- 0,u:CVar <- 9,v:CVar <- 83]^[
    DT:DVar <- 9,caron:DVar <- 1,coolon:DVar <- 0,u:CVar <- 9,v:CVar <- 83]^[
    DT:DVar <- 9,caron:DVar <- 1,coolon:DVar <- 0,u:CVar <- 10,v:CVar <- 84]^[
    DT:DVar <- 10,caron:DVar <- 1,coolon:DVar <- 0,u:CVar <- 10,v:CVar <- 84]^[
    DT:DVar <- 10,caron:DVar <- 0,coolon:DVar <- 0,u:CVar <- 10,v:CVar <- 84]}
    in time 10
```

Figure 5: Automatic transformation and execution in Maude

5.3. Discussion

5.3.1. Approach

As our case study shows, for any cyber-physical systems specified in $CPSL^{sc}$, if the initial state and the program are provided, its automatic simulation and verification can be conducted in Maude. Specifically, according to the form we defined in Maude, we can carry out simulation and verification by rewriting commands with the provided initial state `init`, program `P` and property `sp` as parameters. For any program defined using $CPSL^{sc}$,

- `GCF(P)` converts the program `P` into the guarded choice form.
- `EXE(init:GCF(P))` executes one step of the program `P` in the current data state `init`.
- `autoTransExe(init:P)` simulates the execution of the program `P` in the current data state `init` and returns a trace of data states generated during the execution.
- `trVerify(init:P ^ sp)` checks whether the property `sp` is satisfied during the execution.

5.3.2. Limitation

In this paper, we only use BMS as an example to illustrate the applicability of our proposed approach, and we only focus on the verification of reachability. There are two main limitations of the current approach.

On the one hand, it is not yet possible to analyze complex and large-scale systems. Although this approach can theoretically simulate and verify any systems, the potential state explosion caused by parallel composition (due to the numerous branches it creates) must be addressed. To bridge this gap, the current approach should be supplemented with reduction techniques.

On the other hand, the current approach can only be used to verify reachability. Since there are more complex properties in real CPS, we need to investigate how to verify more complex and general temporal properties.

Maude — maude.darwin64 — 100×17

```
Maude> (trew {trVerify([DT <- 0,caron <- 1,coolon <- 0,u <- 0,v <- 89]: (while DT < 10 do (R(u '= 1)
 until (u >= (DT + 1)) ; if v >= 90 then coolon := 1 else skip ; if v <= 80 then coolon := 0 else sk
ip ; DT := u) ; caron := 0) || (while DT < 10 do (@(caron = 1) ; if coolon == 0 then R(v '= 1) until
 ((coolon = 1 \/ caron = 0) \/ v >= 100) else R(v '= -2) until ((coolon = 0 \/ caron = 0) \/ v <= 0)
))  ^ Exist(v < 80) )} with no time limit .)
rewrites: 19347 in 23ms cpu (24ms real) (813753 rewrites/second)

Timed rewrite  {trVerify([DT:DVar <- 0,caron:DVar <- 1,coolon:DVar <- 0,u:CVar <- 0,v:CVar <- 89]:(
    while DT:DVar < 10 do(R(u:CVar '= 1)until(u:CVar >= DT:DVar + 1); if v:CVar >= 90 then
    coolon:DVar := 1 else skip ; if v:CVar <= 80 then coolon:DVar := 0 else skip ; DT:DVar :=
    u:CVar); caron:DVar := 0)|| while DT:DVar < 10 do(@(caron:DVar = 1); if coolon:DVar == 0 then
    R(v:CVar '= 1)until((coolon:DVar = 1 \/ caron:DVar = 0)\/ v:CVar >= 100)else R(v:CVar '=
    -2)until((coolon:DVar = 0 \/ caron:DVar = 0)\/ v:CVar <= 0))^ Exist(v:CVar < 80))} in
    CPS_SharedV with mode default time increase 1

Result ClockedSystem :
  {NoExistence} in time 10
```

(a) Checking *Exist(v* $\leqslant$ *80)*

Maude — maude.darwin64 — 100×17

```
Maude>  (trew {trVerify([DT <- 0,caron <- 1,coolon <- 0,u <- 0,v <- 89]: (while DT < 10 do (R(u '= 1
) until (u >= (DT + 1)) ; if v >= 90 then coolon := 1 else skip ; if v <= 80 then coolon := 0 else s
kip ; DT := u) ; caron := 0) || (while DT < 10 do (@(caron = 1) ; if coolon == 0 then R(v '= 1) unti
l ((coolon = 1 \/ caron = 0) \/ v >= 100) else R(v '= -2) until ((coolon = 0 \/ caron = 0) \/ v <= 0
)))  ^ Exist(v > 90) )} with no time limit .)
rewrites: 19354 in 24ms cpu (25ms real) (788221 rewrites/second)

Timed rewrite  {trVerify([DT:DVar <- 0,caron:DVar <- 1,coolon:DVar <- 0,u:CVar <- 0,v:CVar <- 89]:(
    while DT:DVar < 10 do(R(u:CVar '= 1)until(u:CVar >= DT:DVar + 1); if v:CVar >= 90 then
    coolon:DVar := 1 else skip ; if v:CVar <= 80 then coolon:DVar := 0 else skip ; DT:DVar :=
    u:CVar); caron:DVar := 0)|| while DT:DVar < 10 do(@(caron:DVar = 1); if coolon:DVar == 0 then
    R(v:CVar '= 1)until((coolon:DVar = 1 \/ caron:DVar = 0)\/ v:CVar >= 100)else R(v:CVar '=
    -2)until((coolon:DVar = 0 \/ caron:DVar = 0)\/ v:CVar <= 0))^ Exist(v:CVar > 90))} in
    CPS_SharedV with mode default time increase 1

Result ClockedSystem :
  {NoExistence} in time 10
```

(b) Checking *Exist(v* $\geqslant$ *90)*

Maude — maude.darwin64 — 100×18

```
Maude> (trew {trVerify([DT <- 0,caron <- 1,coolon <- 0,u <- 0,v <- 89]: (while DT < 10 do (R(u '= 1)
 until (u >= (DT + 1)) ; if v >= 90 then coolon := 1 else skip ; if v <= 80 then coolon := 0 else sk
ip ; DT := u) ; caron := 0) || (while DT < 10 do (@(caron = 1) ; if coolon == 0 then R(v '= 1) until
 ((coolon = 1 \/ caron = 0) \/ v >= 100) else R(v '= -2) until ((coolon = 0 \/ caron = 0) \/ v <= 0)
))  ^ Exist(v <= 88) )} with no time limit .)
rewrites: 13353 in 16ms cpu (16ms real) (817597 rewrites/second)

Timed rewrite  {trVerify([DT:DVar <- 0,caron:DVar <- 1,coolon:DVar <- 0,u:CVar <- 0,v:CVar <- 89]:(
    while DT:DVar < 10 do(R(u:CVar '= 1)until(u:CVar >= DT:DVar + 1); if v:CVar >= 90 then
    coolon:DVar := 1 else skip ; if v:CVar <= 80 then coolon:DVar := 0 else skip ; DT:DVar :=
    u:CVar); caron:DVar := 0)|| while DT:DVar < 10 do(@(caron:DVar = 1); if coolon:DVar == 0 then
    R(v:CVar '= 1)until((coolon:DVar = 1 \/ caron:DVar = 0)\/ v:CVar >= 100)else R(v:CVar '=
    -2)until((coolon:DVar = 0 \/ caron:DVar = 0)\/ v:CVar <= 0))^ Exist(v:CVar <= 88))} in
    CPS_SharedV with mode default time increase 1

Result[ClockedSystem]:
  {[DT:DVar <- 1,caron:DVar <- 1,coolon:DVar <- 1,u:CVar <- 2,v:CVar <- 88]^ FirstExistence} in
    time 2
```

(c) Checking *Exist(v* $\leqslant$ *88)*

Figure 6: Verification Results in Maude

6. Related Work

Recently, as a new form of engineering systems, Cyber-Physical systems (CPS) have gained wide adoption in many fields. However, since CPS are tangled with discrete behaviors of the computer and continuous behaviors of the physical, the behavior of them is complex to formalize. Thus, a number of calculus and languages have been proposed to specify CPS. As an extension of Communicating Sequential Process (CSP) allowing continuous dynamics, Hybrid Communicating Sequential Process (HCSP) [1] was proposed by introducing differential equations to model continuous behaviors and communication interruptions in hybrid systems, and many researches have been carried out around it [14, 15, 16, 17]. He et al. developed a hybrid relational modeling language (HRML) [2] which supports complex combinations of both testing and signal reaction behaviors to model hybrid systems. The means of interaction in HCSP and other process algebras is communication, and communication is structured as a pair of channel and message [1]. Different from the mechanism of them, we proposed a language to specify CPS [3], elaborated its syntax in [4], and we name it $CPSL^{sc}$ in this paper. $CPSL^{sc}$ supports parallel composition, and its interaction is based on shared variables. Particularly, in the syntax of continuous behavior $R(v, \dot{v})$ **until** g, the interaction between the cyber and the physical is achieved through the guard condition g which contains shared variables. In our previous works, we have done a series of research on this language. We proposed its denotational semantics and algebraic semantics [4], developed its proof system [5] based on denotational semantics and extended Hoare triples [18]. Further, we also implemented the transformation from our language to automata in SpaceEx [6, 7]. In this paper, we continue to dive into the semantics of $CPSL^{sc}$. Additionally, simulation and verification based on the proposed semantics are conducted as well.

For a proposed language, on the one hand, a primary concern is giving its formal semantics. With symbols and formulas from a mathematical view, formal semantics can precisely define and interpret the semantics of programming languages. As for defining formal semantics, Hoare and He developed Unifying Theories of Programming (UTP) approach [10], and it has been applied in the formalization of the semantics of various languages [19, 20, 21, 22]. In the UTP approach, three different mathematical models are often used to represent a theory of programming, namely, the operational [23], the denotational [24], and the algebraic [25] approaches. Each of these representations has its distinctive advantages for theories of programming. In our previous work [4], we presented the denotational semantics and algebraic semantics for $CPSL^{sc}$. Since the algebraic semantics is well suited in symbolic calculation of parameters and structures of an optimal design, we focus on the algebraic semantics and elaborate it in this paper based on our previous work.

On the other hand, from an application point of view, formalization and verification for CPS is also worthy of exploration. Banach et al. extended Event-B [26] to Hybrid Event-B [27, 28] which takes account of continuous behaviors, so that CPS can be conducted via B method. Platzer et al. developed a theorem prover called KeYmaera X [29] to verify CPS models. Hybrid automata [30] can also be employed to model and verify CPS, such as formalization and verification conducted via model checkers Uppaal [31, 32] and SpaceEx [33, 34]. In this paper, we apply the rewrite engine Real-Time Maude [8, 9] to mechanize the algebraic semantics of $CPSL^{sc}$ and then implement the simulation and verification for CPS. Real-Time Maude has been widely used in the verification of CPS. James et al. carried out a series of verification for the European Rail Traffic Management System [35, 36] using Real-Time Maude. Meanwhile, as mentioned in [10], "Algebraic proof by term rewriting is a most promising way in which computers can assist in the process of reliable design." Considering that the theoretical basis of Maude lies in rewriting logic [12] and rewriting logic is suitable for giving executable semantics, it matches the gist of our algebraic semantics. Therefore, in this paper, we employ Real-Time Maude to establish the link between algebraic semantics and its application (i.e., simulation and verification). Slightly different from the existing work, our work focuses on the bridge between the algebraic semantics and the corresponding simulation and verification, rather than just verifying properties over CPS. This paper is devoted to exploring the feasibility of simulating and verifying CPS from an algebraic approach, and gives a simple example to showcase its usage.

7. Conclusion and Future Work

In this paper, we have taken the perspective of algebraic semantics to explore an algebraic approach to simulation and verification for Cyber-Physical systems. For this shared variable language proposed to specify CPS, we first elaborated its algebraic semantics, so that programs of $CPSL^{sc}$ can be translated into a unified form called guarded choice form. Moreover, it also indicates that a parallel program can be sequentialized through our algebraic laws.

On this basis, we employed the rewriting engine Real-Time Maude to implement the algebraic laws. Through the mechanization, we can conduct simulation and verification for Cyber-Physical systems with an algebraic approach from a fresh view. To demonstrate the usage of our algebraic semantics and the mechanical implementation in Maude, a battery management system is provided as a case study in this paper.

In the future, we will conduct research from two aspects. Theoretically, we will dive into the semantics linking theory [10] of $CPSL^{sc}$ in tools of formal methods such as Coq [37]. Practically, based on algebraic semantics and combined with reduction techniques, more complex simulation and verification of CPS involving more complicated properties and larger scale systems will be explored.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgments

This work was partially supported by the National Key Research and Development Program of China (No. 2022YFB3305102), the National Natural Science Foundation of China (No. 62032024), the Startup Foundation for Introducing Talent of NUIST (No. 2024r035), the Automatic Software Generation and Intelligent Service Key Laboratory of Sichuan Province (No. CUIT-SAG202306), the "Digital Silk Road" Shanghai International Joint Lab of Trustworthy Intelligent Software (No. 22510750100), Jiangsu Province Engineering Research Center of Advanced Computing and Intelligent Services, and Shanghai Trusted Industry Internet Software Collaborative Innovation Center.

A. Implementation in Maude

A.1. Mechanization of GCF(P||Q)

The following equations are parallel expansion laws implemented in Maude.

```
---(par-1-1), (par-1-2), (par-1-3), (par-1-4), (par-1-5)
ceq GCF(P || Q) = par1(asggc1,Q)[]par1(asggc2,P)
        if asggc1 := GCF(P) /\ asggc2 := GCF(Q) .
ceq GCF(P || Q) = par1(asggc1,Q)[]par2(evtgc2,P)
        if asggc1 := GCF(P) /\ evtgc2 := GCF(Q) .
ceq GCF(P || Q) = par1(asggc1,Q)
        if asggc1 := GCF(P) /\ cbgc2 := GCF(Q) .
ceq GCF(P || Q) = par1(asggc1,Q)[]par1(asggc2,P)[]par2(evtgc2,P)
        if asggc1 := GCF(P) /\ evtgc2[]asggc2 := GCF(Q) .
ceq GCF(P || Q) = par1(asggc1,Q)[]par2(evtgc2,P)
        if asggc1 := GCF(P) /\ evtgc2[]cbgc2 := GCF(Q) .

---(par-2-2), (par-2-3), (par-2-4), (par-2-5)
ceq GCF(P || Q) = par2(evtgc1,Q)[]par2(evtgc2,P)
        if evtgc1 := GCF(P) /\ evtgc2 := GCF(Q) .
ceq GCF(P || Q) = par2(evtgc1,Q)[]par3(cbgc2,P)
        if evtgc1 := GCF(P) /\ cbgc2 := GCF(Q) .
ceq GCF(P || Q) = par1(asggc2,P)[]par2(evtgc1,Q)[]par2(evtgc2,P)
        if evtgc1 := GCF(P) /\ evtgc2[]asggc2 := GCF(Q) .
ceq GCF(P || Q) = par2(evtgc1,Q)[]par2(evtgc2,P)[]par3(cbgc2,P)
        if evtgc1 := GCF(P) /\ evtgc2[]cbgc2 := GCF(Q) .

---(par-3-3), (par-3-4), (par-3-5)
ceq GCF(P || Q) = par33(cbgc1,cbgc2)
        if cbgc1 := GCF(P) /\ cbgc2 := GCF(Q) .
ceq GCF(P || Q) = par1(asggc2,P)[]par2(evtgc2,P)
        if cbgc1 := GCF(P) /\ evtgc2[]asggc2 := GCF(Q) .
ceq GCF(P || Q) = par2(evtgc2,P)[]par33(cbgc1,cbgc2)
        if cbgc1 := GCF(P) /\ evtgc2[]cbgc2 := GCF(Q) .
```

A.2. Mechanization of par*

*par*2 and *par*3 are used to compute the components of parallel composition where one process is an event guarded choice and a cb guarded choice respectively.

```
---par2: One is Event Guarded Choice
eq par2([]{EvtG(evt)-> P'},Q) = []{EvtG(evt)->(P' || Q)} .
eq par2([]{evtg1 -> P'}[]evtgc2,Q) = par2([]{evtg1 -> P'},Q)[]par2(evtgc2,Q) .

---par3: One is Cb Guarded Choice
eq par3([]{CbG(cbeh)-> P'},Q) = []{CbG(cbeh)->(P' || Q)} .
eq par3([]{cbg1 -> P'}[]cbgc2,Q) = par3([]{cbg1 -> P'},Q)[]par3(cbgc2,Q) .
```

B. Operational Semantics

- **Discrete Assignment:** $\langle x := e, \sigma \cdot now = t\rangle \xrightarrow{D=0} \langle \epsilon, \sigma[e/x] \cdot now = t\rangle$

  This rule portrays that if the assignment terminates, then the expression e is evaluated and the value gained is assigned to the variable x. Here, $\sigma[e/x]$ is the same as σ except that the value of x is now associated with the value of e. Here, $D = 0$ represents assignment is an instantaneous transition.

- **Discrete Event Guard:** If $\sigma \vDash gd$, then $\langle @gd, \sigma \cdot now = t\rangle \xrightarrow{D=0} \langle \epsilon, \sigma \cdot now = t\rangle$.

  This rule illustrates that $@gd$ terminates, if gd is satisfied at the initial state σ. Similarly, $D = 0$ indicates this transition costs no time. Otherwise, the rule does not allow us to derive any transition.

- **Continuous Evolution:**
  - **(CE-Evolve):** If $\forall time \in [t, t+D) \bullet (v(time), \sigma \vDash \neg g)$, then

    $\langle R(v, \dot{v})$ **until** $g, \sigma \cdot now = t\rangle \xrightarrow{D>0} \langle R(v, \dot{v})$ **until** $g, \sigma \cdot now = t + D\rangle$.
  - **(CE-Term):** If $v(t), \sigma \vDash g$, then $\langle R(v, \dot{v})$ **until** $g, \sigma \cdot now = t\rangle \xrightarrow{D=0} \langle \epsilon, \sigma \cdot now = t\rangle$.

  (CE-Evolve) explains that the continuous behavior evolves for D time units according to $R(v, \dot{v})$ if g is not triggered within this period. After this transition, now is updated with $t + D$. Here, $v(t), \sigma \vDash \neg g$ means that the current values of continuous variables (recorded as $v(t)$) and discrete variables (recorded in σ) invalidate g. **(CE-Term)** describes the termination of the continuous behavior, and we treat it as instantaneous.

References

[1] Chaochen Zhou, Ji Wang, and Anders P. Ravn. A formal description of hybrid systems. In *Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*, pages 511–530. Springer, 1995.

[2] Jifeng He and Qin Li. A hybrid relational modelling language. In *Concurrency, Security, and Puzzles*, volume 10160 of *Lecture Notes in Computer Science*, pages 124–143. Springer, 2017.

[3] Richard Banach and Huibiao Zhu. Language evolution and healthiness for critical cyber-physical systems. *J. Softw. Evol. Process.*, 33(9), 2021.

[4] Ran Li, Huibiao Zhu, and Richard Banach. Denotational and algebraic semantics for cyber-physical systems. In *ICECCS*, pages 123–132. IEEE, 2022.

[5] Ran Li, Huibiao Zhu, and Richard Banach. A proof system for cyber-physical systems with shared-variable concurrency. In *ICFEM*, volume 13478 of *Lecture Notes in Computer Science*, pages 244–262. Springer, 2022.

[6] Ran Li, Huibiao Zhu, and Richard Banach. Translating CPS with shared-variable concurrency in SpaceEx. In *SETTA*, volume 13649 of *Lecture Notes in Computer Science*, pages 127–133. Springer, 2022.

[7] Ran Li, Huibiao Zhu, and Richard Banach. Translating and verifying cyber–physical systems with shared-variable concurrency in spaceex. *Internet of Things*, 23:100864, 2023.

[8] Peter Csaba Ölveczky and José Meseguer. Semantics and pragmatics of real-time maude. *High. Order Symb. Comput.*, 20(1-2):161–196, 2007.

[9] Peter Csaba Ölveczky. Real-time maude and its applications. In *WRLA*, volume 8663 of *Lecture Notes in Computer Science*, pages 42–79. Springer, 2014.

[10] C. A. R. Hoare and Jifeng He. *Unifying Theories of Programming*. Prentice Hall, Englewood Cliffs, 1998.

[11] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[12] Narciso Martí-Oliet and José Meseguer. Rewriting logic: roadmap and bibliography. *Theor. Comput. Sci.*, 285(2):121–154, 2002.

[13] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebraic Methods Program.*, 78(5):293–303, 2009.

[14] Jiang Liu, Jidong Lv, Zhao Quan, Naijun Zhan, Hengjun Zhao, Chaochen Zhou, and Liang Zou. A calculus for hybrid CSP. In *APLAS*, volume 6461 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2010.

[15] Gaogao Yan, Li Jiao, Shuling Wang, Lingtai Wang, and Naijun Zhan. Automatically generating systemc code from HCSP formal models. *ACM Trans. Softw. Eng. Methodol.*, 29(1):4:1–4:39, 2020.

[16] Shuling Wang, Flemming Nielson, Hanne Riis Nielson, and Naijun Zhan. Modelling and verifying communication failure of hybrid systems in HCSP. *Comput. J.*, 60(8):1111–1130, 2017.

[17] Dimitar P. Guelev, Shuling Wang, and Naijun Zhan. Compositional Hoare-Style reasoning about hybrid CSP in the duration calculus. In *SETTA*, volume 10606 of *Lecture Notes in Computer Science*, pages 110–127. Springer, 2017.

[18] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[19] Huibiao Zhu, Jifeng He, Shengchao Qin, and Phillip J. Brooke. Denotational semantics and its algebraic derivation for an event-driven system-level language. *Formal Aspects Comput.*, 27(1):133–166, 2015.

[20] Ana Cavalcanti, Andy J. Wellings, and Jim Woodcock. The safety-critical java memory model formalised. *Formal Aspects Comput.*, 25(1):37–57, 2013.

[21] Ling Shi, Yongxin Zhao, Yang Liu, Jun Sun, Jin Song Dong, and Shengchao Qin. A UTP semantics for communicating processes with shared variables and its formal encoding in PVS. *Formal Aspects Comput.*, 30(3-4):351–380, 2018.

[22] Feng Sheng, Huibiao Zhu, Jifeng He, Zongyuan Yang, and Jonathan P. Bowen. Theoretical and practical approaches to the denotational semantics for MDESL based on UTP. *Formal Aspects Comput.*, 32(2-3):275–314, 2020.

[23] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebraic Methods Program.*, 60-61:17–139, 2004.

[24] Joseph E. Stoy. Denotational semantics: The scott-strachey approach to programming language theory. 1981.

[25] C. A. R. Hoare, Ian J. Hayes, Jifeng He, Carroll Morgan, A. W. Roscoe, Jeff W. Sanders, Ib Holm Sørensen, J. Michael Spivey, and Bernard Sufrin. Laws of programming. *Commun. ACM*, 30(8):672–686, 1987.

[26] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[27] Richard Banach, Michael J. Butler, Shengchao Qin, Nitika Verma, and Huibiao Zhu. Core hybrid event-b I: single hybrid event-b machines. *Sci. Comput. Program.*, 105:92–123, 2015.

[28] Richard Banach, Michael J. Butler, Shengchao Qin, and Huibiao Zhu. Core hybrid event-b II: multiple cooperating hybrid event-b machines. *Sci. Comput. Program.*, 139:1–35, 2017.

[29] André Platzer and Jan-David Quesel. Keymaera: A hybrid theorem prover for systems (system description). In *IJCAR*, volume 5195 of *Lecture Notes in Computer Science*, pages 171–178. Springer, 2008.

[30] Thomas A. Henzinger. The theory of hybrid automata. In *LICS*, pages 278–292. IEEE Computer Society, 1996.

[31] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In *Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer, 1995.

[32] Jidong Lv, Ehsan Ahmad, and Tao Tang. Non-deterministic delay behavior testing of chinese train control system using UPPAAL-TRON. *IEEE Intell. Transp. Syst. Mag.*, 13(3):58–82, 2021.

[33] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. SpaceEx: Scalable verification of hybrid systems. In *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 379–395. Springer, 2011.

[34] Sanghyun Yoon and Junbeom Yoo. Formal verification of ECML hybrid models with SpaceEx. *Inf. Softw. Technol.*, 92:121–144, 2017.

[35] Phillip James, Andrew Lawrence, Markus Roggenbach, and Monika Seisenberger. Towards safety analysis of ERTMS/ETCS level 2 in real-time maude. In *FTSCS*, volume 596 of *Communications in Computer and Information Science*, pages 103–120. Springer, 2015.

[36] Ulrich Berger, Phillip James, Andrew Lawrence, Markus Roggenbach, and Monika Seisenberger. Verification of the european rail traffic management system in real-time maude. *Sci. Comput. Program.*, 154:61–88, 2018.

[37] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.