Translating and Verifying Cyber-Physical Systems with Shared-Variable Concurrency in SpaceEx

Translating and Verifying Cyber-Physical Systems with Shared-Variable Concurrency in SpaceEx

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

[a]School of Software, Nanjing University of Information Science and Technology, Nanjing, China
[b]Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China
[c]Department of Computer Science, University of Manchester, Manchester, UK

ARTICLE INFO

Keywords:
Cyber-Physical Systems (CPS)
Operational Semantics
Shared Variables
Verification
SpaceEx

ABSTRACT

Cyber-Physical systems (CPS), combining continuous physical behavior and discrete control behavior, have been widely utilized in recent years. However, the traditional modeling languages used to specify discrete systems are no longer applicable to CPS, since CPS subsume the combination of the cyber and the physical. To address this, a modeling language for CPS with shared-variable concurrency is proposed. In this paper, we introduce an extension to the classical configuration of transitions in Structural Operational Semantics (SOS), which adds an auxiliary variable "now" to the data state. Using this configuration, we present operational semantics for this language. Then, we propose a translation strategy from this language to hybrid automata to enable efficient verification for CPS. We give the detailed translation of basic commands and compound constructs formally, and the correctness of this translation is explored as well. To demonstrate the effectiveness of our approach, we provide an example of Autonomous Emergency Braking (AEB) and carry out the corresponding verification using SpaceEx. Compared with the existing work that uses SpaceEx for formal modeling and verification, the translation strategy from programs to automata not only allows any CPS described in this language to be modeled and verified based on the proposed strategy, but also indicates the semantic foundation on which formal verification depends.

1. Introduction

Cyber-Physical systems (CPS) [1] are dynamic systems composed of discrete behaviors of the cyber and continuous behaviors of the physical. In CPS, computer programs can influence physical behaviors, and vice versa. The interdependency and integration between the cyber and the physical are useful in many fields, such as aerospace, automotive, healthcare, manufacturing, and transportation [2].

However, the complexity of this combination can complicate the design of systems. Therefore, it is of primary importance to propose specification languages for CPS. We proposed and elaborated a shared variable language to specify CPS formally in our previous work [3, 4]. Further, a series of studies on the formal semantics of this language were carried out. We explored its denotational semantics and algebraic semantics based on the Unifying Theories of Programming (UTP) approach [5] in [4], and developed its proof system with Hoare logic [6] in [7].

Our previous works mainly concentrate on theoretical semantics. In this paper, we propose operational semantics for this language to enrich its formal semantics. On the other hand, from the practical level, we also implement formal verification based on the operational semantics by translating our language to hybrid automata in SpaceEx. Consequently, for a given CPS scenario, it can be described using our language. Then, formalization and verification can be conducted through our translation.

This paper extends our work published at SETTA 2022 [8]. In our previous work, we mainly focused on the transformation from our modeling language to hybrid automata in SpaceEx [9], and gave partial examples to illustrate this conversion. Now, compared with our previous work, the main contributions of this paper are illustrated in Fig. 1.

- **From Theoretical View:** We propose operational semantics for our language with the aid of Structural Operational Semantics (SOS). In contrast to the classical configuration of a transition in SOS, we add an auxiliary variable *now* to the data state to capture the real-time feature of CPS. Then, we apply transitions to present operational semantics based on the new form of the configuration.

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

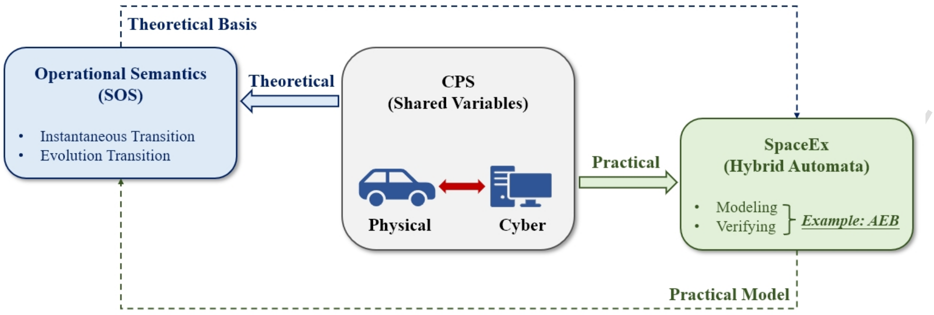


Figure 1: Technology Roadmap

- **From Practical View:** We elaborate the translation from our language to hybrid automata formally, and more examples are appended to this paper as well. Through this transformation, we can build a bridge between our language and hybrid automata in SpaceEx, so that CPS specified by our language can be verified in SpaceEx. Moreover, to illustrate our language and the transformation, we present a case study of Autonomous Emergency Braking (AEB) and verify some properties of AEB.

The remainder of this paper is as follows. In Section 2, we introduce hybrid automata and the model checker SpaceEx, recall the syntax of our modeling language, and discuss the related work. In Section 3, we formalize the operational semantics of this language based on the introduction of transitions. In Section 4, the translation from the language to hybrid automata in SpaceEx is given. Section 5 is dedicated to the case study of AEB. Finally, we conclude our work and discuss some future work in Section 6.

2. Background

In this section, we first briefly introduce hybrid automata and the model checker SpaceEx. Moreover, we recall the syntax of our modeling language to describe CPS.

2.1. Hybrid Automata

We first give the formal definitions of hybrid automata based on [10]. A hybrid automaton can be expressed as a tuple of $A = (Loc, Lab, Edg, Flow, Inv)$.

- Loc stands for a finite set of vertices called locations. A location is associated with an invariant and a flow.
- Lab is a finite set of synchronization labels.
- Edg is a finite set of edges called transitions. By defining transitions, the system can jump between locations. It can be denoted by $e = (l, syn, g, asg, l')$. l and l' represent the source location and the target location respectively. syn is a synchronization label. If $syn = \tau$, it implies that it is an internal transition. g is a guard and asg is an assignment. If the guard of this transition is satisfied, the related assignment can take effect and change the values of variables instantaneously. The synchronization label is used to implement synchronization between different automata.
- $Flow$ contains a set of differential equations that describe the evolution of continuous variables in this location.
- Inv is a function that assigns to each location an invariant. The automaton remains in the current location while the invariant is satisfied.

Then, based on the above introduction, the parallel composition of hybrid automata can be constructed in the following way. Let $A_1 = (Loc_1, Lab_1, Edg_1, Flow_1, Inv_1)$ and $A_2 = (Loc_2, Lab_2, Edg_2, Flow_2, Inv_2)$ be two hybrid automata over the same variable set. The two automata synchronize on the set $Lab_1 \cap Lab_2$, so that once A_1 performs a discrete transition with the synchronization label $syn \in (Lab_1 \cap Lab_2)$, A_2 executes the transition with the same synchronization label syn. We give the formal definition of the parallel composition of $A_1 \times A_2$ as below.

$A_1 \times A_2$ is the form of $(Loc_1 \times Loc_2, Lab_1 \cup Lab_2, Edg, Flow, Inv)$, where

- $((l_1, l_2), syn, g, asg, (l'_1, l'_2)) \in Edg$ iff
  (1) $(l_1, syn_1, g_1, asg_1, l'_1) \in Edg_1$ and $(l_2, syn_2, g_2, asg_2, l'_2) \in Edg_2$,
  (2) $syn_1 = syn_2 = syn$, or $(syn_1 = syn \notin (Lab_1 \cap Lab_2)) \wedge syn_2 = \tau$, or $(syn_2 = syn \notin (Lab_1 \cap Lab_2)) \wedge syn_1 = \tau$
  (3) $g = g_1 \cap g_2$
  (4) $asg = asg_1 \cap asg_2$
- $Flow(l_1, l_2) = Flow_1(l_1) \cap Flow_2(l_2)$
- $Inv(l_1, l_2) = Inv_1(l_1) \cap Inv_2(l_2)$

The operational semantics of hybrid automata is presented below.

- If $(l, syn, g, asg, l') \in Edg$, $v \in g$, $(v, v') \in asg$, $v \in Inv(l)$, $v' \in Inv(l')$, then $\langle l, v\rangle \xrightarrow{syn} \langle l', v'\rangle$.
- If $f \in Flow(l)$, $f(0) = v$, $\forall\, 0 \leqslant t \leqslant D \cdot f(t) \in Inv(l)$, then $\langle l, v\rangle \xrightarrow{D\geqslant 0} \langle l, f(t)\rangle$.

2.2. SpaceEx

SpaceEx is a verification platform for hybrid systems. For a given mathematical model of a hybrid system, it can check whether this model satisfies the desired properties [9, 11]. SpaceEx models are similar to hybrid automata. Moreover, SpaceEx supports hierarchy, templates and instantiations.

A model in SpaceEx contains one or several components. There are two kinds of components:

- Base Component: This is a single hybrid automaton, and it consists of locations and transitions between locations.
- Network Component: This represents a parallel composition of several hybrid automata, and it is comprised of one or more instantiations of other components. By connecting base components via their variables and labels, a network component constructs a parallel composition of base components.

After loading the formalized model into SpaceEx, we can set the initial states, forbidden states, verification scenarios and other options to verify the model. SpaceEx supports three verification scenarios, including Polyhedral Hybrid Automaton Verifier (PHAVer) [12], Le Guernic-Girard (LGG) [9], and Space-Time with Clustering (STC) [13]. In this paper, we adopt the PHAVer scenario to verify our models. PHAVer is applied to Linear Hybrid Automata [14] which are hybrid systems with piecewise constant bounds on the derivatives.

2.3. Syntax of Our Modeling Language

The syntax of our language is summarized in Table 1. This language was proposed in [3] and we elaborated it by detailing the guard conditions of the continuous behaviors in [4]. Here, x is a discrete variable, e is a discrete or continuous expression, and v is a continuous variable. b stands for a Boolean condition, and it contains discrete and continuous variables. A process in our language contains discrete behaviors Db, continuous behaviors Cb, and several compositions and constructs of Db and Cb.

2.3.1. *Discrete Behavior*

This language contains two kinds of discrete behaviors, i.e., discrete assignment $x := e$ and discrete event guard $@gd$.

- $x := e$ is a discrete assignment, which is an atomic action. It evaluates the expression e and assigns the value to the discrete variable x.
- $@gd$ is a discrete event guard. It can be triggered when the discrete guard gd is satisfied. Otherwise, it waits until gd is triggered by the environment. Note that the environment consists of other processes in the parallel composition.

Table 1
Syntax of Our Modeling Language

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

2.3.2. Continuous Behavior

We employ differential relations to describe continuous behaviors in our language.

- $R(v, \dot{v})$ **until** g defines continuous behaviors. It denotes that the continuous variable v evolves as the differential relation $R(v, \dot{v})$ specifies until the guard condition g is met. Four kinds of guard condition g are allowed in our language, including the discrete guard gd, continuous guard gc, mixed guards $gd \wedge gc$ and $gd \vee gc$. Here, in our language, the discrete variables can influence the continuous behaviors through the guard condition g.

2.3.3. Composition

Further, a process can be comprised of the above commands in the following way.

- $P; Q$ is sequential composition. The processes P and Q execute sequentially.
- **if** b **then** P **else** Q is a conditional construct. If the Boolean condition b is true, then the process P will be performed. Otherwise, Q is executed.
- **while** b **do** P is an iteration construct. The process P is executed repeatedly each time the Boolean condition b is true.
- $P \parallel Q$ is parallel composition. It represents that the processes P and Q run in parallel. The parallel mechanism in our language is based on shared variables. In our language, shared writable variables only focus on discrete variables.

2.4. Related Work

Due to the massive proliferation in computer systems that combine with real world, Cyber-Physical systems have been applied in many fields. Therefore, it is tempting to investigate the modeling languages for CPS. A number of languages and calculus have been proposed for specifying CPS. Hybrid CSP (HCSP) [15] is an extension of Communicating Sequential Processes (CSP) by introducing differential equations to model continuous behaviors and communication interruptions in hybrid systems. It supports parallel composition via the communication mechanism. Since HCSP was proposed, lots of research have been conducted on various aspects of HCSP, such as formal semantics [16, 17], formal verification [18, 19], and code generation [20]. He et al. presented a hybrid relational modeling language (HRML) in [21], where a signal-based interaction mechanism is adopted to synchronize activities of hybrid systems. For describing and analysing hybrid systems, Hybrid process algebra (HyPA) which is an extension of the process algebra ACP was proposed in [22]. Additionally, Platzer et al. models hybrid systems with hybrid programs [23, 24] and proposed differential dynamic logic to carry out logical proofs [25, 26], case validation and tool implementation [27, 28] for hybrid systems. We also proposed a language whose parallel mechanism in CPS is based on shared variables in [3], and we simplified its syntax in [4]. Slightly different from [26, 23], our language subdivides the types of variables (i.e., discrete or continuous) for a more realistic description of CPS. In this paper, we conduct research around our shared variable language for CPS.

Moreover, formal semantics can precisely define and interpret the semantics of programming languages with symbols and formulas from a mathematical view. Therefore, a primary concern of a given modeling language is its formal semantics. Unifying Theories of Programming (UTP) approach was developed by Hoare and He in [5], and it has been applied in defining semantics for many languages [29, 30, 31]. It contains three different mathematical models to represent a theory of programming, namely, operational semantics [32], denotational semantics [33] and algebraic semantics [34]. In our previous work [4], based on UTP approach, we explore the denotational semantic and algebraic semantics for our modeling language. In this paper, we propose operational semantics for the shared variable language of CPS to enrich its formal semantics.

Meanwhile, there are many research works on the formalization and verification of CPS. Bu et al. explored online verification of CPS and modeling-verification-fixing framework of event-driven IoT system from bounded reachability analysis of linear hybrid automata [2, 35]. Banach et al. extended Event-B [36] to Hybrid Event-B that includes continuous behavior and discrete transitions for modeling and verifying hybrid systems [37, 38]. Platzer et al. developed a hybrid theorem prover called KeYmaera X to verify hybrid systems [39, 40]. James et al. carried out a series of verification for the European Rail Traffic Management System using Real-Time Maude [41, 42]. In this paper, we employ SpaceEx to realize transformation. SpaceEx [9] has been applied in formalization and verification in many fields [43, 44, 11]. Aman et al. established a relationship between rTIMO networks and a class of timed safety automata in [45]. Inspired by this, in this paper, we translate our modeling language of CPS to the hybrid automata in SpaceEx. Consequently, any CPS specified by our language can be formalized and verified in SpaceEx according to our transformation method.

3. Operational Semantics of Our Language

In this section, we introduce transitions used in our operational semantics, and then we present the detailed transition rules.

3.1. Transition

A classical configuration in a transition is formalized as $\langle P, \sigma \rangle$, where P and σ stand for the process and the data state of *only the discrete variables* in P.

For example, if $P =_{df} x := 1; y := y + 2$ and $\sigma =_{df} \{x \mapsto 0, y \mapsto 0\}$, the configuration $\langle P, \sigma \rangle$ represents that the current data state is $x = 0 \land y = 0$, and the program to be executed in the current state is a sequential composition $x := 1; y := y + 2$.

Based on the definition of configuration, a transition describes how the process moves from one configuration to the next configuration. A transition is in the form of $\langle P, \sigma \rangle \rightarrow \langle P', \sigma' \rangle$. It indicates that executing the program P one step in the current state σ can lead to a new state σ' and P' is the remainder of P.

Therefore, for the above example, we have:

$\langle x := 1; y := y + 2, \sigma \rangle \rightarrow \langle y := 1, \sigma_1 \rangle \rightarrow \langle \epsilon, \sigma_2 \rangle$, where $\sigma_1 =_{df} \{x \mapsto 1, y \mapsto 0\}$, $\sigma_2 =_{df} \{x \mapsto 1, y \mapsto 2\}$ and ϵ stands for an empty process.

Slightly different from the classical structures, in addition to discrete variables of the process involved in σ, we also introduce a new continuous variable called *now* to represent the global clock. The configuration in our transitions is the form of $\langle P, \sigma \cdot now = t \rangle$. Moreover, if execution reaches a configuration $\langle \epsilon, \sigma' \cdot now = t' \rangle$, it indicates that the process terminates in the state σ' at the time t'.

Besides, for continuous variables in the process P, since their values are from dense types, there is no need to record their values in σ. In this paper, for simplicity, we assume that the differential relation $R(v, \dot{v})$ can be transformed into a function of the continuous variable v with respect to the time t. Then, we can obtain their values by reference to the time (i.e., $v(t)$ means v's value at the time t).

A transition rule for our language has the form of $\langle P, \sigma \cdot now = t \rangle \xrightarrow{D \geqslant 0} \langle P', \sigma' \cdot now = t + D \rangle$. Here, D is a real number (equal to or greater than 0) which stands for the duration of the transition.

- If $D = 0$, it stands for an instantaneous transition. It means that the process P executes one step in the current state σ, and then the process changes to P' which stands for the remainder of P and the state is now updated to σ'. This transition is instantaneous and costs no time, so *now* keeps unchanged.
- If $D > 0$, it is an evolution transition and it can portray the evolution of continuous behaviors. After this evolution, *now* is updated to $t + D$.

Table 2
Operational Semantics

Discrete Assignment:	$\langle x := e, \sigma \cdot now = t\rangle \xrightarrow{D=0} \langle \epsilon, \sigma[e/x] \cdot now = t\rangle$
Discrete Event Guard:	If $\sigma \vDash gd$, then $\langle @gd, \sigma \cdot now = t\rangle \xrightarrow{D=0} \langle \epsilon, \sigma \cdot now = t\rangle$.
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** $g, \sigma \cdot now = t\rangle \xrightarrow{D=0} \langle \epsilon, \sigma \cdot now = t\rangle$.
Sequential Composition:	
(SC-NTerm)	If $\langle P, \sigma \cdot now = t\rangle \xrightarrow{D\geqslant 0} \langle P', \sigma' \cdot now = t+D\rangle$,
	then $\langle P;Q, \sigma\rangle \xrightarrow{D\geqslant 0} \langle P';Q, \sigma' \cdot now = t+D\rangle$.
(SC-Term)	If $\langle P, \sigma \cdot now = t\rangle \xrightarrow{D\geqslant 0} \langle \epsilon, \sigma' \cdot now = t+D\rangle$,
	then $\langle P;Q, \sigma \cdot now = t\rangle \xrightarrow{D\geqslant 0} \langle Q, \sigma' \cdot now = t+D\rangle$.
Conditional Construct:	
(CC-If)	If $v(t), \sigma \vDash b$, then $\langle$**if** b **then** P **else** $Q, \sigma \cdot now = t\rangle \xrightarrow{D=0} \langle P, \sigma \cdot now = t\rangle$.
(CC-Else)	If $v(t), \sigma \vDash \neg b$, then $\langle$**if** b **then** P **else** $Q, \sigma \cdot now = t\rangle \xrightarrow{D=0} \langle Q, \sigma \cdot now = t\rangle$.
Iteration Construct:	
(IC-Loop)	If $v(t), \sigma \vDash b$, then $\langle$**while** b **do** $P, \sigma \cdot now = t\rangle \xrightarrow{D=0} \langle P;$**while** b **do** $P, \sigma \cdot now = t\rangle$.
(IC-Term)	If $v(t), \sigma \vDash \neg b$, then $\langle$**while** b **do** $P, \sigma \cdot now = t\rangle \xrightarrow{D=0} \langle \epsilon, \sigma \cdot now = t\rangle$.
Parallel Composition:	
(PC-Dist)	If $\langle P, \sigma \cdot now = t\rangle \xrightarrow{D=0} \langle P', \sigma' \cdot now = t\rangle$,
	then $\langle P \parallel Q, \sigma \cdot now = t\rangle \xrightarrow{D=0} \langle P' \parallel Q, \sigma' \cdot now = t\rangle$,
	$\langle Q \parallel P, \sigma \cdot now = t\rangle \xrightarrow{D=0} \langle Q \parallel P', \sigma' \cdot now = t\rangle$.
(PC-Cont)	If $\langle R1, \sigma \cdot now = t\rangle \xrightarrow{D>0} \langle R1, \sigma \cdot now = t+D\rangle$,
	$\langle R2, \sigma \cdot now = t\rangle \xrightarrow{D>0} \langle R2, \sigma \cdot now = t+D\rangle$,
	then $\langle R1 \parallel R2, \sigma \cdot now = t\rangle \xrightarrow{D>0} \langle R1 \parallel R2, \sigma \cdot now = t+D\rangle$,
	$\langle R2 \parallel R1, \sigma \cdot now = t\rangle \xrightarrow{D>0} \langle R2 \parallel R1, \sigma \cdot now = t+D\rangle$.
(PC-Term)	If $\langle P, \sigma \cdot now = t\rangle \xrightarrow{D\geqslant 0} \langle \epsilon, \sigma' \cdot now = t+D\rangle$,
	then $\langle P \parallel Q, \sigma \cdot now = t\rangle \xrightarrow{D\geqslant 0} \langle Q, \sigma' \cdot now = t+D\rangle$,
	$\langle Q \parallel P, \sigma \cdot now = t\rangle \xrightarrow{D\geqslant 0} \langle Q, \sigma' \cdot now = t+D\rangle$.

3.2. Transition Rules

As enumerated in Table 2, we employ the following transition rules to present the operational semantics of our language.

3.2.1. Discrete Behavior Semantics

- Discrete Assignment: 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.

- Discrete Event Guard: This rule illustrates that $@gd$ terminates, if gd is satisfied at the initial state σ. Otherwise, the rule does not allow us to derive any transition.

3.2.2. Continuous Behavior Semantics

- Continuous Evolution: **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.

3.2.3. Composition Semantics

Based on the operational semantics of the above discrete and continuous behaviors, we give the semantics for the composition of basic commands.

- Sequential Composition: These rules depict that the process first activates P, and when P terminates then Q is executed.
- Conditional Construct: These rules represent that the process performs P if b is *true*. Otherwise, Q is selected instead.
- Iteration Construct: Analogously, if b is *true*, the process executes the loop body process P and then determines whether the Boolean condition b is satisfied again. The iteration construct ends when b is *false*.
- Parallel Composition: For parallel composition, σ and σ' are global states.
  - **PC-Dist** means that if one of the parallel components performs an instantaneous transition, the whole parallel process also makes this transition while leaving its partner unchanged. More specifically, if the parallel components are both discrete behaviors, they perform as interleaving. If one is discrete and the other is continuous, then the discrete one is assumed to be scheduled first.
  - **PC-Cont** describes the parallel composition of two continuous behavior. When both $R1$ and $R2$ evolve for D time units, $R1 \parallel R2$ evolves for D time units as well.
  - **PC-Term** indicates if one of the parallel components terminates (i.e., reaches an empty process ϵ), then the whole parallel composition is left with its partner.

4. Translation in SpaceEx

In this section, for a given process P of our language, we translate it to the form of a hybrid automaton $A = (Loc, Lab, Edg, Flow, Inv)$. The components in the automaton are updated depending on the detailed structure of the process P, and we present the detailed transformation of basic commands and compound constructs in turn.

4.1. Variables

As the foundation of the transformation, we first describe how to define variables in SpaceEx and introduce some vital variables that we used in our transformation.

4.1.1. Discrete Variables and Continuous Variables

In our language, there are discrete variables and continuous variables. There are only continuous variables (local or global) and constants in SpaceEx. Thus, to define discrete variables in SpaceEx, we can consider them as a special kind of continuous variables whose derivative is always 0.

4.1.2. Crucial Variables

In our transformation, a global clock variable needs to be defined, so that it captures the real-time feature of CPS. Therefore, we define a continuous variable t whose meaning is quite similar to *now* in the above semantics. t is controlled by a $Clock$ automaton that simulates the real time clock. The detailed formalization of $Clock$ is shown in Section 5.2.2.

Additionally, we define $tert$ as a local variable controlled by the respective independent automaton. We treat $tert$ as a discrete variable, so we set $tert' == 0$ in all locations. $tert$ is used to realize the instantaneous jump in the automata, thereby ensuring the correctness of the simulation results returned by SpaceEx.

4.2. Discrete Behavior

For discrete behaviors, there are two statements in our language, including discrete assignment $x := e$ and discrete event guard $@gd$.

4.2.1. Discrete Assignment

As introduced in Section 2.2, the edges of the graph can allow the system to jump between locations [9]. It can change the values of variables with the assignment. Hence, we can simply realize the discrete assignment by adding the corresponding assignment statement to the edge. If $P = x := e$, we have:

- $Loc = Loc \cup \{term\}$
- $Edg = Edg \cup \{(init, change \vee \tau, \ , x := e, term)\}$
- $Flow(init) = Flow(term) = \{x' == 0\&tert' == 0\}$
- $Inv(init) = \{tert == t\}$

As for the locations, the *init* location is the source location of this assignment, and it represents the location where the existing automaton (i.e., automaton converted from the previous process) runs. If there is no existing automaton, we start the translation with a bare initial location $init$. We add a new location called *term* to represent the terminal location after this assignment.

The newly added edge is applied to realize the assignment. For its synchronization label, if x is a shared variable, we set the label as $change$ to synchronize this assignment with other processes. Otherwise, the label is τ, indicating this transition is an internal transition. In SpaceEx, a transition without a label name is an internal transition.

Since x and $tert$ are both discrete variables, their derivatives are 0 in all locations. We assume $init$ is an instantaneous location, so we set $Inv(init)$ as $tert == t$.

4.2.2. Discrete Event Guard

For discrete event guard, it can be triggered by the process itself or by the environment. In this formalization, we apply a synchronization label $change$ to let the process observe the environment's action. Note that the observation through the label $change$ means that the process can perceive all changes on shared variables, no matter whether this change can really trigger gd. To formalize the behavior of $@gd$, we have:

- $Loc = Loc \cup \{wait, im, term\}$
- $Edg = Edg \cup \{(init, \tau, gd, \ , term)\} \cup \{(init, \tau, \neg gd, \ , wait)\} \cup \{(wait, change, \ , tert := t, im)\}$ $\cup \{(im, \tau, \neg gd, \ , wait)\} \cup \{(im, \tau, gd, \ , term)\}$
- For all $l \in Loc$, $Flow(l) = \{tert' == 0\}$
- $Inv(init) = Inv(im) = \{tert == t\}$, $Inv(wait) = \{\neg gd\}$

Here, $init$ and $term$ are the initial and terminal locations of $@gd$. The $wait$ location represents that gd has not been triggered and the process is waiting for the environment. We introduce the intermediate location im to determine whether the newly changed value by the environment can trigger gd.

For the edges, according to the initial data state, the automaton of $@gd$ moves from the $init$ location to the $term$ location or the $wait$ location. As mentioned before, the trigger action can be done by the process itself (i.e., gd is satisfied at the $init$ location) or by the environment (i.e., the environment changes the corresponding variables and thus triggers gd). If the initial state cannot activate gd, the automaton jumps from the $init$ location to the $wait$ location. The automaton stays stuck in this location until the environment changes the variables in gd, and then reaches the im location. If gd is satisfied at the im location, the process moves to the $term$ location. Otherwise, it returns to the $wait$ location and waits for the environment again.

Similarly, we set $tert$'s derivative as 0 in all locations. Moreover, since x cannot be changed during the execution of $@gd$, no restrictions of x are necessary to add to the flow of locations in the automaton of $@gd$. Same as the $init$ location, the im location is instantaneous and we set $Inv(im)$ as $tert == t$. When the automaton stays at the $wait$ location where gd is not satisfied, thus $Inv(wait) = \{\neg gd\}$.

Example 1. We take $@x > 1$ as an example to illustrate the detailed formalization of $@gd$, and Fig. 2 presents its automaton in SpaceEx.

Here, x is the shared variable controlled by the environment. It can be changed by the environment and these changes can be perceived by $@gd$.

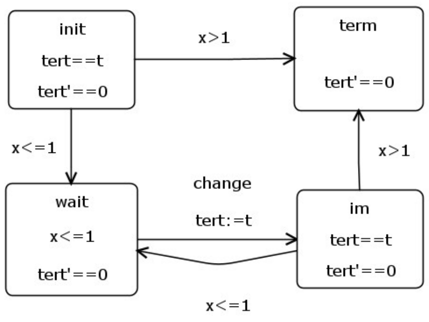


Figure 2: $@(x > 1)$

For $@gd$, it moves from the *init* location to the *term* location, if gd is satisfied (i.e., $x > 1$) at the beginning. If the current data state cannot meet gd (i.e., $x \leqslant 1$), the process jumps to the *wait* location where the process waits for the environment to change x. Once the environment changes x, the environment automaton synchronizes with the $@x > 1$ automaton through the *change* label. Consequently, the automaton reaches the *im* location. Further, it moves to the *term* location if the current value of x meets $x > 1$. Otherwise, the automaton returns to the *wait* location.

$\square$

4.3. Continuous Behavior

For the continuous behavior $R(v, \dot{v})$ **until** g, we formalize the models according to the types of the guard g, including gc, gd, $gd \vee gc$ and $gd \wedge gc$.

4.3.1. $g \equiv gc$

If the guard condition is purely determined by continuous variables, the evolution of the continuous variable v follows the differential relation until gc is satisfied. Thus, the process is not concerned with the discrete behaviors of the environment. For $P = R(v, \dot{v})$ **until** gc, we have:

- $Loc = Loc \cup \{evolve, term\}$
- $Edg = Edg \cup \{(init, \tau, gc, \ , term)\} \cup \{(init, \tau, \neg gc, \ , evolve)\} \cup \{(evolve, \tau, gc, tert := t, term)\}$
- $Flow(init) = Flow(term) = \{tert' == 0 \& v' == 0\}, Flow(evolve) = \{tert' == 0 \& R(v, \dot{v})\}$
- $Inv(init) = \{tert == t\}$, $Inv(evolve) = \{\neg gc\}$

Similar to the *wait* location in the $@gd$ automaton, the *evolve* location implies that gc has not been triggered. The difference between the *wait* location and the *evolve* location is that differential relation $R(v, \dot{v})$ is added in the flow of the *evolve* location, i.e., $Flow(evolve) = \{tert' == 0 \& R(v, \dot{v})\}$. When the automaton is in the *evolve* location, it means that the continuous behavior is evolving as the differential relation specifies. Once gc is satisfied, the automaton moves to the *term* location which indicates the continuous behavior terminates. Due to the termination, it requires that $Flow(term) = \{tert' == 0 \& v' == 0\}$.

Example 2. As presented in Fig. 3, we take $\dot{v} = 1$ **until** $v \geqslant 10$ as an instance to show this transformation in SpaceEx.

Here, v is a continuous variable controlled by the automaton of $\dot{v} = 1$ **until** $v \geqslant 10$. If the initial state meets $v \geqslant 10$, the automaton reaches the *term* location at once. If not, it moves to the *evolve* location where the differential relation $\dot{v} == 1$ is accompanied. During this evolution, once $v \geqslant 10$ is satisfied, the process terminates and the automaton jumps to the *term* location.

Besides, it is worth noting that we set $Inv(evolve) = \{v \geqslant 10\}$ rather than $\{v < 10\}$ (i.e., $\{\neg gc\}$ in the above formal definition) in this automaton. This is because of the feature of hybrid automata. If we set the invariant as $\{v < 10\}$, it can never jump to the *term* location. For simplicity, in this paper, we express the invariant as $\{\neg gc\}$ in formal definition, but the real automaton incorporates the boundary value into the invariant.

$\square$

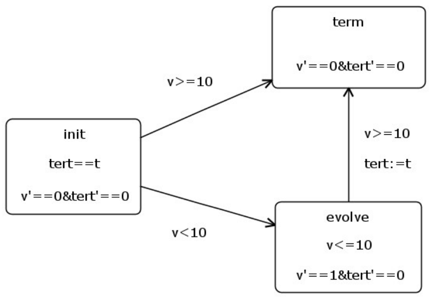


Figure 3: $\dot{v} = 1$ **until** $v \geqslant 10$

4.3.2. $g \equiv gd$

When the guard condition involves merely discrete variables, the process is evolving until gd is triggered. Since gd can be triggered by the environment, changes of gd by the environment should be noticed. For $P = R(v, \dot{v})$ **until** gd, we have:

- $Loc = Loc \cup \{evolve, im, term\}$
- $Edg = Edg \cup \{(init, \tau, gd, \ , term)\} \cup \{(init, \tau, \neg gd, \ , evolve)\} \cup \{(evolve, change, \ , tert := t, im)\}$ $\cup \{(im, \tau, \neg gd, \ , evolve)\} \cup \{(im, \tau, gd, \ , term)\}$
- $Flow(init) = Flow(im) = Flow(term) = \{tert' == 0\&v' == 0\}$, $Flow(evolve) = \{tert' == 0\&R(v, \dot{v})\}$
- $Inv(init) = Inv(im) = \{tert == t\}$, $Inv(evolve) = \{\neg gd\}$

The construction of $R(v, \dot{v})$ **until** gd is quite similar to the way $@gd$ is formalized. The $wait$ location in the $@gd$ automaton is replaced by the $evolve$ location here in the $R(v, \dot{v})$ **until** gd automaton. The $evolve$ location indicates that gd has not been activated and the continuous behavior is performing.
Example 3. Fig. 4 describes the automaton of $\dot{v} = 1$ **until** $x > 1$ in SpaceEx.

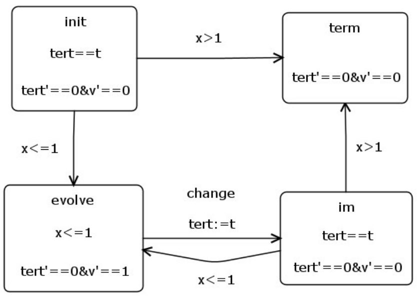


Figure 4: $\dot{v} = 1$ **until** $x > 1$

The meanings of these locations and transitions among them are the same as those in the $@x > 1$ automaton (shown in Fig. 2), except that $\dot{v} == 1$ is appended in the flow of the $evolve$ location.

□

4.3.3. $g \equiv gd \vee gc$

If the guard condition is a hybrid one with the form of $gd \vee gc$, the process evolves until gd or gc is satisfied. As a result, we need to pay attention not only to when the evolution of the process makes gc hold, but also to when the behaviors of the environment make gd hold. For $P = R(v, \dot{v})$ **until** $gd \vee gc$, we have:

- $Loc = Loc \cup \{evolve, im, term\}$
- $Edg = Edg \cup \{(init, \tau, gd, \ , term)\} \cup \{(init, \tau, gc, \ , term)\} \cup \{(init, \tau, \neg gd \& \neg gc, \ , evolve)\}$ $\cup \{(evolve, \tau, gc, tert := t, term)\} \cup \{(evolve, change, \ , tert := t, im)\} \cup \{(im, \tau, \neg gd, \ , evolve)\}$ $\cup \{(im, \tau, gd, \ , term)\}$
- $Flow(init) = Flow(im) = Flow(term) = \{tert' == 0 \& v' == 0\}$, $Flow(evolve) = \{tert' == 0 \& R(v, \dot{v})\}$
- $Inv(init) = Inv(im) = \{tert == t\}$, $Inv(evolve) = \{\neg gd \& \neg gc\}$

The added locations and transitions have similar meanings as those introduced before, so we omit the detailed explanation here. When the automaton is at the *evolve* location, it means that neither gd nor gc is satisfied.

Example 4. $\dot{v} = 1$ **until** $x > 1 \vee v \geqslant 10$ is employed as an example and the corresponding model is given in Fig. 5.

If the initial state meets $x > 1$ or $v \geqslant 10$, the process terminates and the automaton jumps from the *init* location to the *term* location. Otherwise, it implies neither gd nor gc can be triggered. Then, the automaton reaches the *evolve* location where the continuous variable v evolves as $\dot{v} == 1$.

During this evolution, as soon as $v \geqslant 10$ is satisfied, the automaton reaches the *term* location. Also, during this period, once the environment changes the value of x, the automaton runs into the *im* location and checks whether the newly updated value of x caters to $x > 1$. The flow of waiting and triggering gd is consistent with the previous description of $@gd$.

□

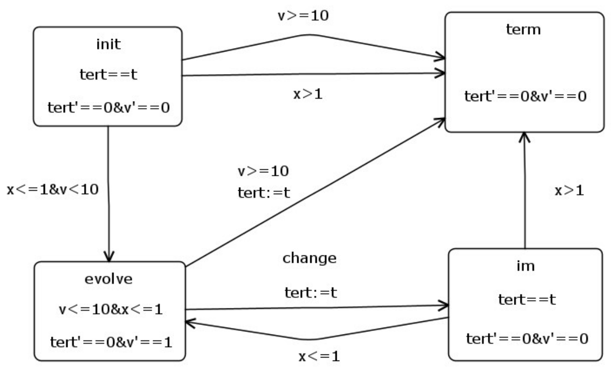


Figure 5: $\dot{v} = 1$ **until** $x > 1 \vee v \geqslant 10$

4.3.4. $g \equiv gd \wedge gc$

In this situation, only when gd and gc are both satisfied, the relevant continuous behavior terminates. Thus, in this condition, we also need to focus on the process itself and the environment as well. We construct the model of $R(v, \dot{v})$ **until** $gd \wedge gc$ as below.

- $Loc = Loc \cup \{evolve1, evolve2, im, term\}$
- $Edg = Edg \cup \{(init, \tau, gd \& gc, \ , term)\} \cup \{(init, \tau, \neg gd \& gc, \ , evolve1)\} \cup \{(init, \tau, \neg gc, \ , evolve2)\}$ $\cup \{(evolve1, change, \ , tert := t, \ im)\} \cup \{(evolve1, \tau, \neg gc, \ , evolve2)\} \cup \{(evolve2, \tau, \neg gd \& gc, \ , evolve1)\}$ $\cup \{(evolve2, \tau, gd \& gc, tert := t, term)\} \cup \{(evolve2, change, \ , tert := t, im)\} \cup \{(im, \tau, \neg gd \& gc, \ , evolve1)\}$ $\cup \{(im, \tau, \neg gc, \ , evolve2)\} \cup \{(im, \tau, gd \& gc, \ , term)\}$
- $Flow(init) = Flow(im) = Flow(term) = \{tert' == 0 \& v' == 0\}$, $Flow(evolve1) = Flow(evolve2) = \{tert' == 0 \& R(v, \dot{v})\}$
- $Inv(init) = Inv(im) = \{tert == t\}$, $Inv(evolve1) = \{\neg gd \& gc\}$, $Inv(evolve2) = \{\neg gc\}$

Since SpaceEx cannot support $\vee$, just one evolving location fails to depict $\neg gd \vee \neg gc$. Therefore, we set two evolving locations. In these two locations, the process is evolving according to the differential relation. The difference between these two locations is that the process enters the $evolve1$ location when gc holds but gd is not satisfied, and moves to the $evolve2$ location if gc does not hold. Here, when the automaton arrives at the $evolve2$ location, whether gd holds is not our concern. That is, the $evolve2$ location contains two situations, i.e., neither gd nor gc is satisfied, and gd holds but gc is unsatisfied. Considering that changes on gd from the environment need to be noticed, we introduce this intermediate location in a similar way as before.

Example 5. More specifically, we take the process of $\dot{v} = 1$ **until** $x > 1 \wedge v \geqslant 10$ as an example to explain this formalization in detail. Fig. 6 is its automaton.

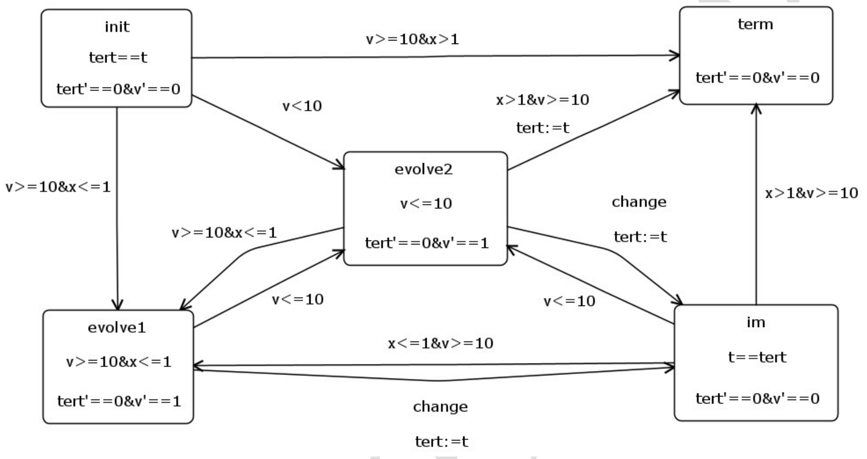


Figure 6: $\dot{v} = 1$ **until** $x > 1 \wedge v \geqslant 10$

If the data state meets $x > 1 \wedge v \geqslant 10$ at the beginning, the automaton enters the $term$ location which indicates that the evolution ends. Otherwise, the process is evolving as the differential relation specifies. We abstract the evolution state into two locations.

If $v \geqslant 10$ is satisfied at the beginning, the process moves to the $evolve1$ location where the process only needs to wait for triggering $x > 1$. During this evolution, the environment can change the value of x and synchronize this change via the label $change$. Further, the automation reaches the im location. If $v \geqslant 10 \wedge x > 1$, the automaton jumps to the $term$ location. If $v \geqslant 10 \wedge x \leqslant 1$, the automaton returns the $evolve1$ location. If $v \leqslant 10$, the automaton goes from the im location to the $evolve2$ location.

On the other hand, if $v \geqslant 10$ is unsatisfied at the beginning, the automaton moves to the $evolve2$ location. In this location, $x > 1$ can or cannot be triggered. Similarly, once the environment changes x, the automaton can move to the im location during the period in the $evolve2$ location. Depending on the value of v, there is a shift between the $evolve1$ location and the $evolve2$ location.

□

4.4. Composition

Based on the models of discrete behaviors and continuous behaviors, we now translate the composition of the above commands into models in SpaceEx.

4.4.1. Sequential Composition

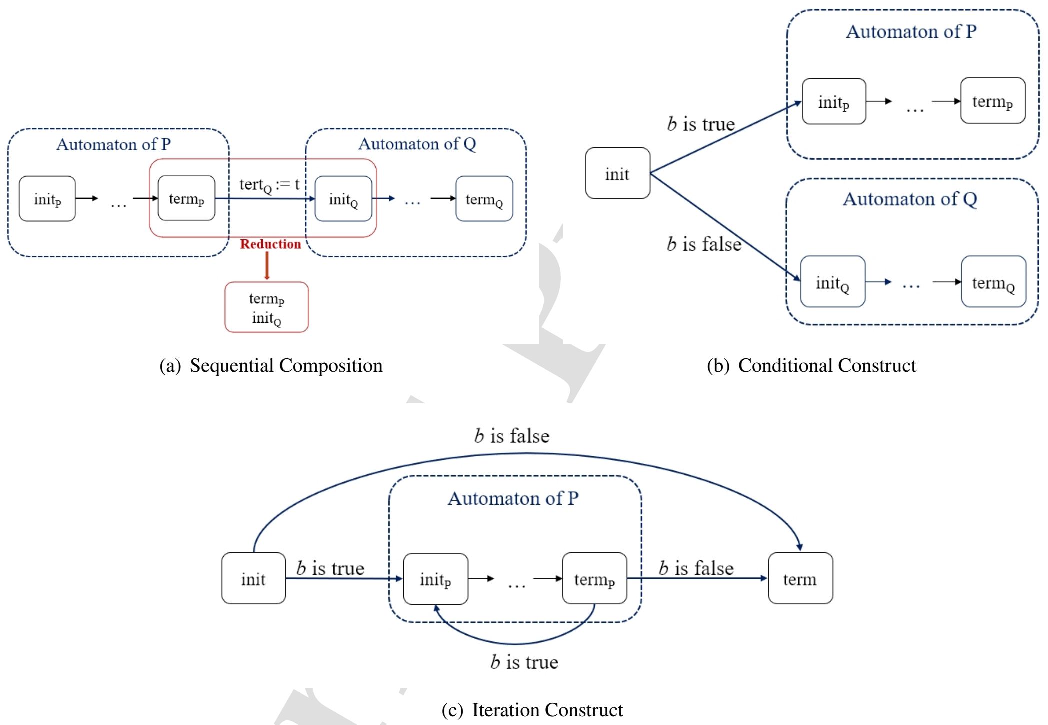
For the sequential composition $P; Q$, as shown in Fig. 7(a), we can simply connect the two automata P and Q with a transition $(term_P, \tau, \ , tert_Q := t, init_Q)$. This transition is from P's terminal location to Q's initial location, and it assigns the current time (i.e., P's terminal time) to the initial value of $tert_Q$ (i.e., Q's initial time). Further, we can also combine them into one location for reduction.

4.4.2. Conditional Construct

As for conditional construct **if** b **then** P **else** Q, we need to determine whether to execute P or Q. As illustrated in Fig. 7(b), if the Boolean condition b is *true* in the current state (i.e., in the *init* location), then P is selected to execute. Otherwise, Q is executed. We connect the *init* location to the initial location of the corresponding process to be executed (i.e., $init_P$ or $init_Q$) with a transition whose guard is b or $\neg b$. Formally, to construct the model of conditional construct, transitions $(init, \tau, b, \ , init_P)$ and $(init, \tau, \neg b, \ , init_Q)$ are added.

4.4.3. Iteration Construct

For iteration construct **while** b **do** P, we present its construction in Fig. 7(c). If the Boolean condition b is *false* at the very beginning (i.e., in the *init* location), the process terminates at once without executing P. Consequently, the automaton moves from the *init* location to the *term* location through the transition $(init, \tau, \neg b, \ , term)$. If b is *true*, with the transition $(init, \tau, b, \ , init_P)$, P will be executed repeatedly. We accomplish the loop by adding a transition $(term_P, \tau, b, \ , init_P)$ from the $term_P$ location to the $init_P$ location. After executing P several times, if b is *false*, the automaton can jump out of the loop and enter the *term* location through the transition $(term_p, \tau, \neg b, \ , term)$.



(a) Sequential Composition (b) Conditional Construct

(c) Iteration Construct

Figure 7: Models of Compositions in SpaceEx

4.4.4. Parallel Composition

SpaceEx provides convenience for us to construct the automaton of parallel composition. As we introduced before, network components in SpaceEx support the connection of base components in parallel. Thus, instead of formalizing the automaton manually, we can first construct automata for parallel components (in their respective base components), and then just connect them as a whole parallel process (in the network component) by binding the variables (i.e., shared variables) and labels of these parallel components.

4.5. Correctness of the Translation

To explore the correctness of the translation from the processes of our language to the hybrid automata, a simulation relation $\looparrowright$ is defined, based on their operational semantics.

Definition 1. Simulation Relation
A simulation relation $\looparrowright$ over processes of our language and the hybrid automata is a weak unidirectional simulation, if whenever $\langle P, \sigma \cdot now = t\rangle \looparrowright (A_P, \langle l_P, v_P\rangle)$:

If $\langle P, \sigma \cdot now = t\rangle \xrightarrow{D\geq 0} \langle P', \sigma' \cdot now = t'\rangle$, then $\langle l_P, v_P\rangle \rightarrow^* \langle l'_P, v'_P\rangle$ and $\langle P', \sigma' \cdot now = t'\rangle \looparrowright (A_P, \langle l'_P, v'_P\rangle)$.

For data states of the process and its converted automaton, there is equivalence between them, that is, $(\sigma \cdot now = t) = v_P$ and $(\sigma' \cdot now = t') = v'_P$.

□

Here, $\rightarrow^*$ stands for several steps of transition rules defined in the operational semantics of hybrid automata in Section 2. Then, the definition of simulation relation can be lifted to $P \looparrowright A_P$, iff for all states $\sigma \cdot now = t$ in P and $\langle l_P, v_P\rangle$ in A_P, $\langle P, \sigma \cdot now = t\rangle \looparrowright (A_P, \langle l_P, v_P\rangle)$ is satisfied. Having defined this simulation relation, we can state our theorem as follows.

Theorem 1. *For a given process of our language P, we have: $P \looparrowright A_P$.*

Proof. We use the structural induction to prove the above theorem, and we take the parallel composition as an example. If $P \looparrowright A_P$ and $Q \looparrowright A_Q$, then their parallel composition $P \parallel Q$ and its corresponding automaton $A_{P\parallel Q}$ should also satisfy simulation relation, i.e., $P \parallel Q \looparrowright A_{P\parallel Q}$. For simplicity, we assume that P is going to perform an assignment $x := e$, where x is a shared variable of P and Q. Then, we analyze the subsequent changes after this assignment.

- For the process P, it performs an assignment $x := e$:
  From its operational semantics, we have: $\langle P, \sigma \cdot now = t\rangle \xrightarrow{D=0} \langle P', \sigma' \cdot now = t\rangle$, where $\sigma' = \sigma[e/x]$.
  From the translation and the operational semantics of hybrid automaton, we have: $\langle l_P, v\rangle \xrightarrow{change} \langle l'_P, v'\rangle$, where $v' = v[e/x]$.

- For the process Q, nothing is done:
  From its operational semantics, since we assume that the result of the parallel composition is done by interleaving, so Q does nothing and the configuration keeps unchanged.
  From its corresponding hybrid automaton, we have: $\langle l_Q, v\rangle \xrightarrow{change \vee \tau} \langle l'_Q, v\rangle$. Here, if Q needs to synchronize with P at the current location l_Q, the automaton will jump to the next location l'_Q through a transition labeled as $change$. Otherwise, it stays at the current location (i.e., $l'_Q = l_Q$) with a τ transition.

- For the parallel composition $P \parallel Q$, it appears as doing an assignment $x := e$:
  From its operational semantics, we have: $\langle P \parallel Q, \sigma \cdot now = t\rangle \xrightarrow{D=0} \langle P' \parallel Q, \sigma' \cdot now = t\rangle$, where $\sigma' = \sigma[e/x]$.
  From the translation and the operational semantics of hybrid automaton, we have: $\langle (l_P, l_Q), v\rangle \xrightarrow{change} \langle (l'_P, l'_Q), v'\rangle$, where $v' = v[e/x]$.
  Therefore, we have: $P \parallel Q \looparrowright A_{P\parallel Q}$.

This theorem indicates the correctness of the translation from the processes of our language to the hybrid automata.

5. Case Study

The work closest to this paper is the work of Aman and Ciobanu, they established a relationship between rTIMO networks and a class of timed safety automata in [45]. They provided the corresponding automata translation strategy based on the structure of the program, and adapted the TravelShop example to perform verification in Uppaal. Their work focuses on mobile systems that communicate through channels, while we conduct transformation and verification for CPS with shared-variable concurrency. Based on our proposed translation strategy in Section 4, we take the process of Autonomous Emergency Braking (AEB) as a case study to showcase the usage of our language and the proposed translation strategy.

In this section, we first give an overview of the process of AEB. Then, with the guidance of the above formalization, we construct its model in the form of automata and verify some related properties in SpaceEx. AEB is abstracted from the real-world problem and it is easy to understand, and other programs can be transformed in the same way. In this section, we only use AEB as an example to illustrate the applicability of our proposed approach.

5.1. Overview

Nowadays, many vehicles are equipped with AEB. AEB is used to improve vehicle safety by allowing the vehicle to warn or apply braking directly when an emergency happens.

As illustrated in Fig. 8, we only focus on the process of braking for simplicity. We assume that the car is waiting for the warning of AEB. Then, on the condition that the driver notices the warning and responds with the brake, the car decelerates uniformly with the acceleration of -4 until the velocity v equals to 0. Otherwise, AEB begins to brake the car sightly and the car decelerates with the acceleration of -2. This phase of deceleration evolves until the velocity equals to 0 or AEB increases the braking force. After two seconds of the warning, if the driver ignores the warning message, AEB increases the braking force and brakes the car with the acceleration of -5.

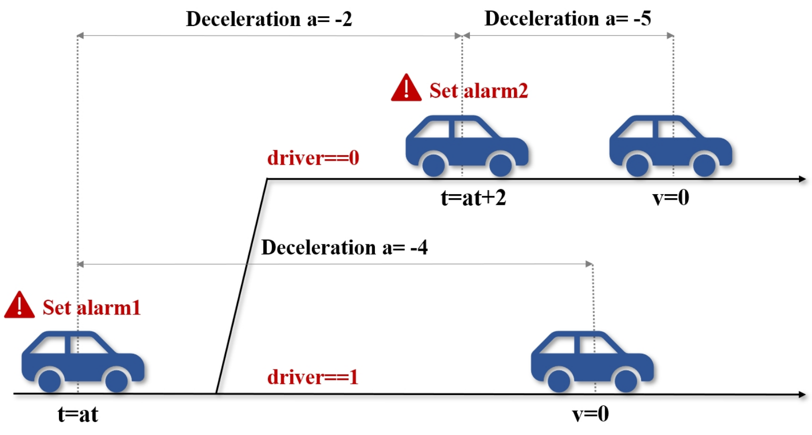


Figure 8: Illustration of AEB

We describe the above simplified example using our language as below. Here, the whole system Sys is the parallel composition of $Clock$, AEB and Car.

$$
\begin{aligned}
&Sys =_{df}\ Clock \parallel AEB \parallel Car;\\
&Clock =_{df}\ \dot{t} = 1\ \textbf{until}\ false;\\
&AEB =_{df}\ alarm1 := 1; at := t; \dot{wt} = 1\ \textbf{until}\ t \geqslant at + 2;\\
&\qquad\qquad \textbf{if}\ (driver == 0)\ \textbf{then}\ \{\ alarm2 := 1;\ \}\\
&Car =_{df}\ @(alarm1 = 1);\\
&\qquad\qquad \textbf{if}\ (driver == 1)\ \textbf{then}\ \{\ v' == -4\ \textbf{until}\ v = 0;\ \}\\
&\qquad\qquad \textbf{else}\ \left\{ \begin{array}{l} v' = -2\ \textbf{until}\ (alarm2 = 1 \lor v = 0);\\ v' = -5\ \textbf{until}\ v = 0; \end{array} \right\}
\end{aligned}
$$

- $Clock$ is defined to simulate the global clock. The continuous variable t stands for the global clock in this system.
- Car defines the behaviors of the vehicle. In this process, the continuous variable v is the current velocity of the car. $alarm1$ and $alarm2$ are discrete variables and represent warnings from the AEB. The discrete variable $driver$ denotes whether the driver responds to the warning of $alarm1$. If $driver = 1$, it means that the driver is conscious and reacts to the warning. Otherwise, it implies that the driver ignores the warning and therefore the car will wait for the control from AEB.

- AEB represents the control of autonomous braking. In this process, wt is an auxiliary variable which is used to realize the delay operation. at is a discrete variable defined to record when AEB sounds $alarm1$. If AEB makes the warning, the corresponding alarm (i.e., $alarm1$ and $alarm2$) equals 1. Otherwise, the values of $alarm1$ and $alarm2$ are assigned to 0.

5.2. Formalization

In this subsection, we employ the above program to show how a Cyber-Physical system expressed using our language can be transformed into automata in SpaceEx with the translation approach described previously.

5.2.1. *Outline of Formalization*

The overall approach to the transformation of this example in SpaceEx is ***from bottom to up***. More concretely, since the whole system Sys is comprised of three parallel components (i.e, $Clock$, AEB and Car), we first formalize these parallel components as three base components in SpaceEx and then connect them into a network component Sys. Furthermore, for these parallel components, we convert them to the models in SpaceEx step by step. According to the previous introduction of formalization for basic commands, we translate these individual basic commands. Then, under the guidance of composition in Section 4.4, we link them and form a complete model.

5.2.2. *Base Component of Clock*

Actually, we have mentioned the $Clock$ automaton in Section 4.1. As presented in Fig. 9, the base component $Clock$ has only one continuous variable t. t is controlled by the $Clock$ automaton and it cannot be changed by any other automata. t is evolving as the flow $t' == 1$ defined and we regard it as a global clock.

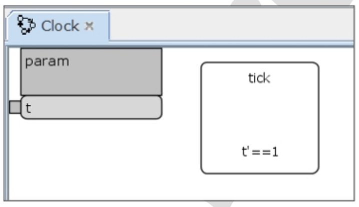


Figure 9: Base Component of $Clock$

5.2.3. *Base Component of AEB*

Fig. 10 shows the automaton of AEB. In this model, we set wt and at as local variables. t, $driver$, $alarm1$ and $alarm2$ are global variables. In addition, a local discrete variable $tert$ is considered as an auxiliary variable (mentioned in Section 4.1). Among these variables, wt, at, $alarm1$, $alarm2$ and $tert$ are controlled by AEB.

Now, we translate the process into automata in SpaceEx. The beginning of AEB is two assignments, we use the $init$ location and the $setAlarm1$ location to formalize it. The transition labeled with $change1$ realizes the two assignments. Here, $change1$ is used to synchronize AEB and Car, so that the change on the shared variable $alarm1$ can be observed by the Car immediately. Next, the $setAlarm1$ location, the $wait2sec$ location and the $termAlarm1$ location are modeled to construct $\dot{wt} = 1$ **until** $t \geqslant at + 2$ in a similar way as before. The process of AEB ends with a conditional construct, so two transitions (i.e., one represents the explicit if branch and the other stands for the omitted $else$ branch) are added from the $termAlarm1$ location to the $term$ location.

5.2.4. *Base Component of Car*

The automaton of Car is illustrated in Fig. 11. In this model, v, $driver$, t, $alarm1$ and $alarm2$ are global variables, and the first three are controlled variables. Similarly, $tert$ is a local auxiliary variable and is controlled by Car.

The beginning of the process Car is a discrete event guard. The $init$ location, the $waitAlarm1$ location, the $imAlarm1$ location and the $recAlarm1$ location are defined to formalize this command. As introduced before, the synchronization label $change1$ can inform Car of the change on the variable $alarm1$. With the label, if the environment (i.e., AEB) changes the value of $alarm1$, the automaton switches from the $waitAlarm1$ location to the $imAlarm1$ location.

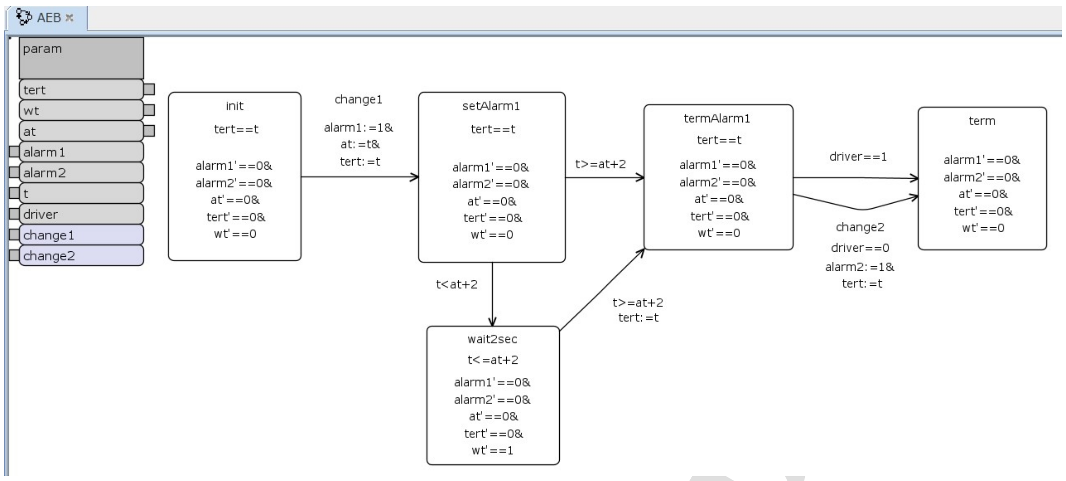


Figure 10: Base Component of AEB

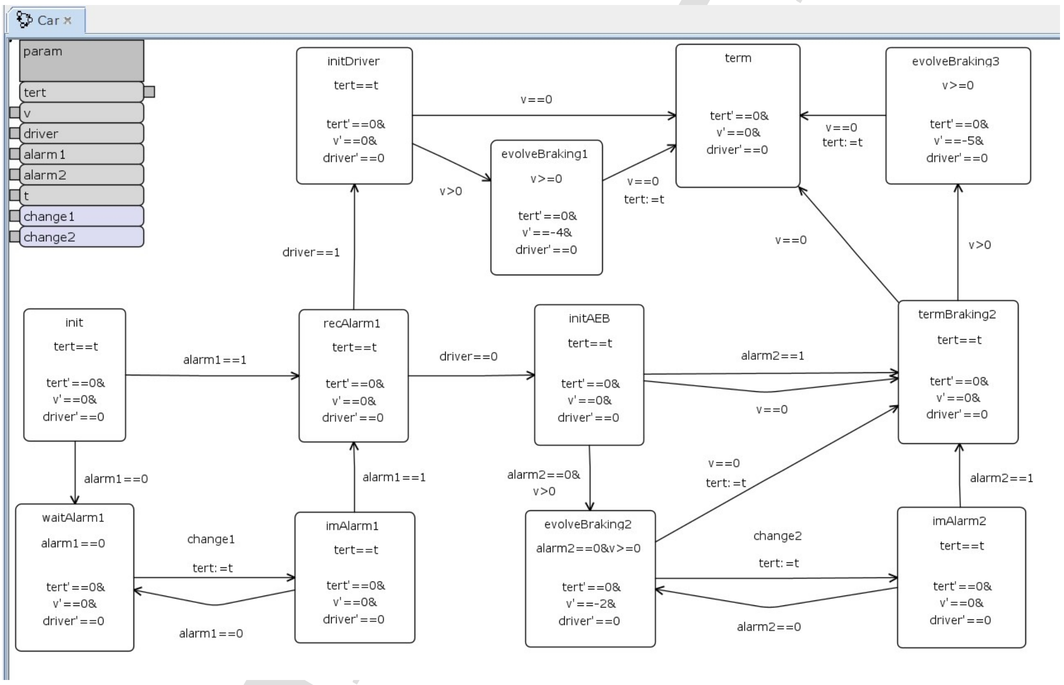


Figure 11: Base Component of Car

Afterwards, it is a conditional construct. Depending on whether the Boolean condition $driver == 1$ holds, we define two transitions. Thus, if $driver == 1$, the automaton reaches the $initDriver$ location. Otherwise, it enters the $initAEB$ location.

If $driver == 1$, it indicates that the driver responds to the alarm and then the car brakes with an acceleration of -4 until its velocity drops to 0. It is a statement with the form of $R(v, \dot{v})$ **until** gc, so we use the $initDriver$ location, the $evolveBraking1$ location and the $term$ location to depict this behavior.

On the contrary, if $driver == 0$, the AEB will adopt measures. The car moves with an acceleration of -2 until its velocity becomes 0 or the AEB sounds $alarm2$. Four locations are applied to portray this behavior, including the

$initAEB$ location, the $evolveBraking2$ location, the $imAlarm2$ location and the $termBraking2$ location. Here, Car synchronizes with AEB through the synchronization label $change2$. After reaching the $termBraking2$ location, the acceleration of the car becomes -5 until its velocity equals 0. In addition to this location, the $evolveBraking3$ location and the $term$ location are used to formalize this behavior.

5.2.5. *Network Component of Sys*

After constructing the above parallel components (i.e., $Clock$, AEB and Car), we give the parallel composition of them. Fig. 12 is a network component for the system. It contains three base components and we connect them via their variables and labels.

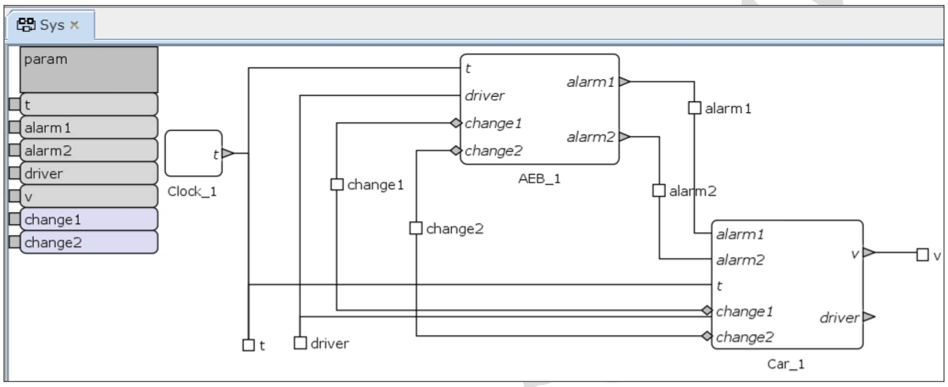


Figure 12: Network Component of Sys

5.3. Verification

In this subsection, we employ the model checker of SpaceEx to verify several properties of the model constructed above.

5.3.1. *Setting*

We now perform verification using SpaceEx. We simulate two scenarios (i.e., $driver = 1$ and $driver = 0$) to check three properties (i.e., *Termination*, *VelocityLimit* and *Evolve*). In this paper, we adopt PHAVer [12] as the verification scenario and we need to manually set the initial states and forbidden states.

Scenario 1 depicts that the driver pays attention to the warning, and Scenario 2 stands for the condition where AEB starts automatic braking when the driver ignores the warning. We set the initial states of the two scenarios as below.

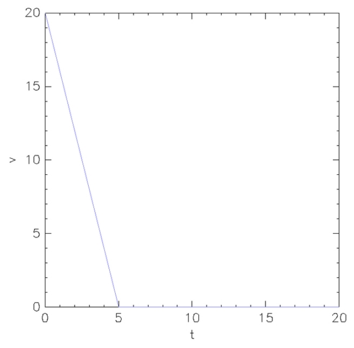
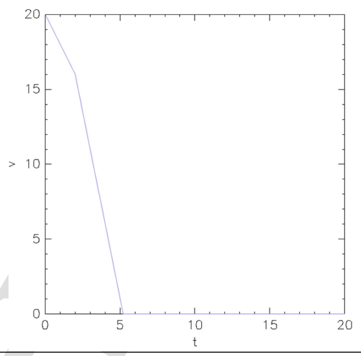
- **Scenario 1:**
  $t == 0 \;\&\; wt == 0 \;\&\; at == 0 \;\&\; AEB\_1.tert == 0 \;\&\; alarm1 == 0 \;\&\; alarm2 == 0 \;\&\; driver == 1$ $\&\; Car\_1.tert == 0 \;\&\; v == 20 \;\&\; loc(AEB\_1) == init \;\&\; loc(Car\_1) == init$

- **Scenario 2:**
  $t == 0 \;\&\; wt == 0 \;\&\; at == 0 \;\&\; AEB\_1.tert == 0 \;\&\; alarm1 == 0 \;\&\; alarm2 == 0 \;\&\; driver == 0$ $\&\; Car\_1.tert == 0 \;\&\; v == 20 \;\&\; loc(AEB\_1) == init \;\&\; loc(Car\_1) == init$

We perform verification for two safety properties (i.e., Property 1 and Property 2) and one reachability property (i.e., Property 3) using SpaceEx.

- **Property 1** is *Termination*; when the car reaches the $term$ location, its velocity must be 0. We set the forbidden states as $loc(Car_1) == term \;\&\; v > 0$.
- **Property 2** is *VelocityLimit*; the velocity must always be in the range 0 to 20. The forbidden states of this property are defined as $v < 0 \mid v > 20$.
- **Property 3** is *Evolve*; we define this property to show the evolution of velocity.

Table 3
Verification Results

Property	Format	Results in Scenario1	Results in Scenario2
1: *Termination*	Text(txt)	{ }	{ }
2: *VelocityLimit*	Text(txt)	{ }	{ }
3: *Evolve*	2D(gen)		

5.3.2. Verification Results

The conducted verification has yielded several positive outcomes, which are detailed in Table 3.

- Property 1 and Property 2 return the verification result { }. It indicates that the system meets the properties of *Termination* and *VelocityLimit*, as forbidden states are not reachable.
- Verification results of Property 3 are 2D graph outputs of SpaceEx. The returned two figures display the evolution of the velocity v over time t and they meet our expectations.

This case study showcases the use of our language and the practicality of our transformation. The proposed transformation strategy allows the utilization of SpaceEx's model checking abilities to verify the properties of CPS described in our language. Meanwhile, the verification carried out on the case study of AEB also validates our proposed operational semantics.

6. Conclusion and Future Work

In [4], we elaborated the language suitable for modeling Cyber-Physical systems, based on the previous work [3]. In this paper, we presented its operational semantics by extending the classical configuration of transitions. Then, we established a bridge between our language and hybrid automata through a translation, enabling the verification of CPS specified with our language using SpaceEx. Finally, we demonstrated the practicality of our approach by showcasing its use in modeling and verifying Autonomous Emergency Braking (AEB) which is abstracted from a real-world scenario.

From the theoretical view, since we proposed the algebraic and denotational semantics of this language in our previous work [4], we will apply the Unifying Theories of Programming (UTP) approach [5] to investigate the link among operational semantics, denotational semantics and algebraic semantics of this language. From the practical view, the automatic translation of our language to models in SpaceEx will be explored, so that more complex CPS can be transformed and verified in a more convenient and automated way. In addition, we also plan to extend our language to support more features relevant to CPS, such as probabilistic behavior.

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

Not applicable.

Acknowledgments

This work was partially supported by the National Key Research and Development Program of China (No. 2022 YFB3305102), the National Natural Science Foundation of China (Grant No. 62032024), the "Digital Silk Road" Shanghai International Joint Lab of Trustworthy Intelligent Software (No. 22510750100), Shanghai Trusted Industry Internet Software Collaborative Innovation Center, and the Dean's Fund of Shanghai Key Laboratory of Trustworthy Computing (East China Normal University).

References

[1] Ruggero Lanotte, Massimo Merro, and Simone Tini. A probabilistic calculus of cyber-physical systems. *Inf. Comput.*, 279:104618, 2021.

[2] Lei Bu, Jiawan Wang, Yuming Wu, and Xuandong Li. From bounded reachability analysis of linear hybrid automata to verification of industrial CPS and IoT. In *SETSS*, volume 12154 of *Lecture Notes in Computer Science*, pages 10–43. Springer, 2019.

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

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

[7] 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.

[8] 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.

[9] 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.

[10] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Theor. Comput. Sci.*, 138(1):3–34, 1995.

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

[12] Goran Frehse. Phaver: algorithmic verification of hybrid systems past hytech. *Int. J. Softw. Tools Technol. Transf.*, 10(3):263–279, 2008.

[13] Goran Frehse, Rajat Kateja, and Colas Le Guernic. Flowpipe approximation and clustering in space-time. In *HSCC*, pages 203–212. ACM, 2013.

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

[15] 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.

[16] 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.

[17] Xiong Xu, Jean-Pierre Talpin, Shuling Wang, Bohua Zhan, and Naijun Zhan. Semantics foundation for cyber-physical systems using higher-order UTP. *ACM Trans. Softw. Eng. Methodol.*, 32(1):9:1–9:48, 2023.

[18] 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.

[19] Xiong Xu, Shuling Wang, Bohua Zhan, Xiangyu Jin, Jean-Pierre Talpin, and Naijun Zhan. Unified graphical co-modeling, analysis and verification of cyber-physical systems by combining AADL and simulink/stateflow. *Theor. Comput. Sci.*, 903:1–25, 2022.

[20] 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.

[21] 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.

[22] Pieter J. L. Cuijpers and Michel A. Reniers. Hybrid process algebra. *J. Log. Algebraic Methods Program.*, 62(2):191–245, 2005.

[23] André Platzer. Differential dynamic logic for hybrid systems. *J. Autom. Reason.*, 41(2):143–189, 2008.

[24] André Platzer. Correction to: Differential dynamic logic for hybrid systems. *J. Autom. Reason.*, 66(1):173, 2022.

[25] André Platzer. Differential logic for reasoning about hybrid systems. In *HSCC*, volume 4416 of *Lecture Notes in Computer Science*, pages 746–749. Springer, 2007.

[26] Simon Lunel, Stefan Mitsch, Benoît Boyer, and Jean-Pierre Talpin. Parallel composition and modular verification of computer controlled systems in differential dynamic logic. In *FM*, volume 11800 of *Lecture Notes in Computer Science*, pages 354–370. Springer, 2019.

[27] Timm Liebrenz, Paula Herber, and Sabine Glesner. Deductive verification of hybrid control systems modeled in simulink with keymaera X. In *ICFEM*, volume 11232 of *Lecture Notes in Computer Science*, pages 89–105. Springer, 2018.

[28] Jan-David Quesel, Stefan Mitsch, Sarah M. Loos, Nikos Aréchiga, and André Platzer. Correction to: How to model and prove hybrid systems with keymaera: a tutorial on safety. *Int. J. Softw. Tools Technol. Transf.*, 23(5):827, 2021.

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

[30] 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.

[31] Feng Sheng, Huibiao Zhu, Jifeng He, Zongyuan Yang, and Jonathan P. Bowen. Theoretical and practical aspects of linking operational and algebraic semantics for MDESL. *ACM Trans. Softw. Eng. Methodol.*, 28(3):14:1–14:46, 2019.

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

[33] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[34] 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.

[35] Yuming Wu, Lei Bu, Jiawan Wang, Xinyue Ren, Wen Xiong, and Xuandong Li. Mixed semantics guided layered bounded reachability analysis of compositional linear hybrid automata. In *VMCAI*, volume 13182 of *Lecture Notes in Computer Science*, pages 473–495. Springer, 2022.

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

[37] 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.

[38] 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.

[39] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. Keymaera X: an axiomatic tactical theorem prover for hybrid systems. In *CADE*, volume 9195 of *Lecture Notes in Computer Science*, pages 527–538. Springer, 2015.

[40] Jan-David Quesel, Stefan Mitsch, Sarah M. Loos, Nikos Aréchiga, and André Platzer. How to model and prove hybrid systems with keymaera: a tutorial on safety. *Int. J. Softw. Tools Technol. Transf.*, 18(1):67–91, 2016.

[41] 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.

[42] 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.

[43] Huixing Fang, Jianqi Shi, Huibiao Zhu, Jian Guo, Kim Guldstrand Larsen, and Alexandre David. Formal verification and simulation for platform screen doors and collision avoidance in subway control systems. *Int. J. Softw. Tools Technol. Transf.*, 16(4):339–361, 2014.

[44] Stefano Minopoli and Goran Frehse. SL2SX translator: From simulink to spaceex models. In *HSCC*, pages 93–98. ACM, 2016.

[45] Bogdan Aman and Gabriel Ciobanu. Real-time migration properties of rtimo verified in uppaal. In *SEFM*, volume 8137 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2013.