Translating CPS with Shared-Variable Concurrency in SpaceEx

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

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

Abstract. Cyber-physical 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 However, the traditional modeling languages used to specify discrete systems are no longer applicable to CPS, since CPS subsume the combination of the cyber and the physical.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 based on shared variables is proposed. In this paper, we present an implementation of this language in SpaceEx. Thus, a bridge between our language and hybrid automata is established.

Keywords: Cyber-physical System (CPS) · Hybrid Automata · SpaceEx.

1 Introduction

Cyber-physical systems (CPS) [5] are dynamical 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 is useful in many fields, such as aerospace, automotive, healthcare and manufacturing [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. For instance, Hybrid CSP (HCSP) [8] is an extension of Communicating Sequential Processes (CSP) by introducing differential equations in hybrid systems. He et al. presented a hybrid relational modeling language (HRML) in [4], where a signal-based interaction mechanism is adopted to synchronize activities of hybrid systems. In contrast, we proposed a language whose parallel mechanism in CPS is based on shared variables, [1]. We provided its denotational semantics and algebraic semantics in [6], and developed its proof system in [7]. Based on our previous work [1,6], in this paper, we connect the language with hybrid automata in SpaceEx. 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.

This work was partially supported by the National Natural Science Foundation of China (Grant Nos. 61872145, 62032024).

The remainder of this paper is as follows. In Section 2, we introduce the model checker SpaceEx and recall the syntax of our modeling language. In Section 3, the translation from the language to models in SpaceEx is given. Finally, we conclude our work and discuss some future work in Section 4.

2 Background

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

2.1 SpaceEx

SpaceEx [3] is a verification platform for hybrid systems. A model in SpaceEx contains one or several components. There are two kinds of components: **Base Component** (a single hybrid automaton) and **Network Component** (a parallel composition of several hybrid automata).

To be specific, in a base component, a vertex is called a location. A location is associated with an invariant and a flow. The automaton remains in the current location while the invariant is satisfied. A flow contains a set of differential equations that describe the evolution of continuous variables in this location. The edges between locations are called transitions. By defining transitions, the system can jump between locations. A transition can be associated with a synchronization label, a guard and an assignment. If the guard of this transition is satisfied, the related assignment can take effect and thus changes the values of variables instantaneously. The synchronization label is used to implement synchronization between different automata. By connecting base components via their variables and labels, a network component constructs a parallel composition of base components.

2.2 Syntax of Our Modeling Language

The syntax of our language is summarized in Table 1. This language was proposed in our previous work [1] and we elaborated it by detailing the guard conditions of the continuous behaviors in [6]. Here, x is a discrete variable, e is a discrete or continuous expression, v is a continuous variable and b stands for a Boolean condition.

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. Here, the environment consists of the other programs 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	$gc ::= true \mid v = e \mid v < e \mid v > e \mid gc \vee gc \mid gc \wedge gc \mid \neg gc$	

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 discrete guard gd, continuous guard gc, mixed guards $gd \wedge gc$ and $gd \vee gc$.

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, and the parallel mechanism is based on shared variables.

3 Translation in SpaceEx

In this section, we convert our language to the form of hybrid automata in SpaceEx. We explain how to define variables. Then, we present the detailed transformation of basic commands and compound constructs in turn.

3.1 Variables

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

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

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 which is controlled by a *Clock* automaton that simulates the real time clock. Moreover, we define *tert* as a local discrete variable controlled by the respective independent automaton. The terminal value of *tert* stands for the time when the program terminates.

3.2 Discrete behavior

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

Discrete Assignment. As introduced in Subsection 2.1, the edges of the graph can allow the system to jump between locations [3]. It can change values of variables with the assignment. Hence, we can simply realize the discrete assignment by adding the corresponding assignment statement to the edge.

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

- *init*: It is the initial location of $@gd$. One special point is that we set all initial states as instantaneous in our model. According to the initial data state, the automaton of $@gd$ moves from the *init* location to the *term* location or the *wait* location.
- *term*: When the discrete guard gd is triggered, the program runs to the terminate location *term*. As mentioned before, the trigger action can be done by the program itself (i.e., gd is satisfied at the *init* location) or by the environment (i.e., the environment changes the corresponding variables and triggers gd).
- *wait*: This location represents that gd has not been triggered and the program is waiting for the environment. 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.
- *im*: We introduce this intermediate location to determine whether the newly changed value by the environment can trigger gd. If gd is satisfied, the program moves to the *term* location. Otherwise, it returns to the *wait* location and waits for the environment again.

Example 1. We take $@x > 1$ as an example to illustrate the detailed formalization of $@gd$, and Fig. 1(a) 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 $@gd$. As introduced in Subsection 3.1, t is a global continuous variable which represents the global clock. *tert* is a local discrete variable and it records when $@x > 1$ terminates. 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 at 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.

3.3 Continous behavior

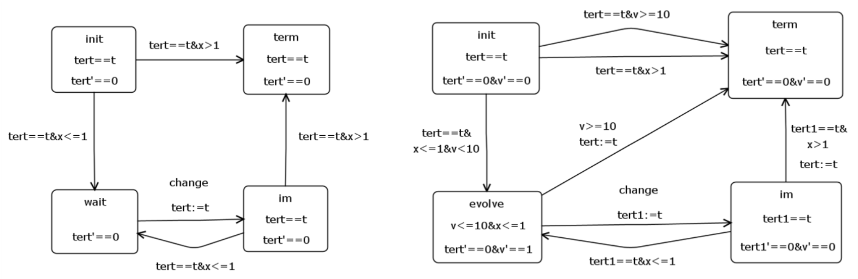
For the continuous behavior $R(v, \dot{v})$ **until** g, we formalize the models according to the types of the guard g, including gc, gd, $gd \vee gc$ and $gd \wedge gc$. Due to the space limitations, we take $R(v, \dot{v})$ **until** $gd \vee gc$ as an example.

$\boldsymbol{g \equiv gd \vee gc}$**.** If the guard condition is a hybrid one with the form of $gd \vee gc$, the program evolves until gd or gc is satisfied. As a result, we need to pay attention not only to when the evolution of the program makes gc hold, but also to when the behavior of the environment makes gd hold. Four locations (i.e., *init*, *evolve*, *im* and *term*) are defined to portray this statement.

- *init*: It stands for the initial location. As mentioned before, we assume that it is an instantaneous location and nothing needs to change at this location.
- *evolve*: Similar to the *wait* location in the $@gd$ automaton, the *evolve* location implies that neither gd nor gc is satisfied. When the automaton is in the *evolve* location, it means that the continuous behavior is evolving as the differential relation specifies.
- *im*: Considering that changes on gd from the environment need to be noticed, we introduce this intermediate location in a similar way as before.
- *term*: Once gd or gc is satisfied, the automaton moves to this location which indicates the continuous behavior terminates.

Example 2. $\dot{v} = 1$ **until** $x > 1 \vee v \geqslant 10$ is employed as an example and the corresponding model is given in Fig. 1(b).

If the initial state meets $x > 1$ or $v \geqslant 10$, the program 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 and the terminal time *tert* is assigned to the current time point. Also, during this period, once the environment changes x, the automaton runs into the *im* location and checks whether the newly updated value of x caters to $x > 1$.



(a) Example 1. $@(x > 1)$ (b) Example 2. $\dot{v} = 1$ **until** $x > 1 \vee v \geqslant 10$

Fig. 1. Models of Examples in SpaceEx

3.4 Composition

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

For the ***sequential composition*** $P; Q$, we can simply connect the two automata P and Q with a transition. This transition is from P's terminal location to Q's initial location, and it assigns P's terminal time to Q's initial time.

As for the ***conditional construct*** **if** b **then** P **else** Q, we need to determine whether to execute P or Q. 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 program (i.e., P or Q) to be executed with a transition whose guard is b or $\neg b$.

For the ***iteration construct*** **while** b **do** P, 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. If b is true, P will be executed repeatedly until b is false. We accomplish it by adding a transition from P's terminal location to P's initial location. After executing P several times, if b is false, the automaton can jump out the loop and enter the *term* location.

For the ***parallel composition*** $P \parallel Q$, we can first construct automata for parallel components (in their respective base components) and then connect them as a whole parallel program (in the network component).

4 Conclusion and Future Work

In [6], we elaborated our language for cyber-physical systems, based on our previous work [1]. In this paper, we transformed this language into automata in SpaceEx [3]. Therefore, under the guidance of the conversion, any CPS specified by our language can be modeled and verified in SpaceEx. In the future, the automatic translation of our language to models in SpaceEx will be explored.

References

1. Banach, R., Zhu, H.: Language evolution and healthiness for critical cyber-physical systems. J. Softw. Evol. Process. **33**(9) (2021)
2. Bu, L., Wang, J., Wu, Y., Li, X.: From bounded reachability analysis of linear hybrid automata to verification of industrial CPS and iot. In: SETSS. Lecture Notes in Computer Science, vol. 12154, pp. 10–43. Springer (2019)
3. Frehse, G., Guernic, C.L., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: Spaceex: Scalable verification of hybrid systems. In: CAV. Lecture Notes in Computer Science, vol. 6806, pp. 379–395. Springer (2011)
4. He, J., Li, Q.: A hybrid relational modelling language. In: Concurrency, Security, and Puzzles. Lecture Notes in Computer Science, vol. 10160, pp. 124–143. Springer (2017)
5. Lanotte, R., Merro, M., Tini, S.: A probabilistic calculus of cyber-physical systems. Inf. Comput. **279**, 104618 (2021)
6. Li, R., Zhu, H., Banach, R.: Denotational and algebraic semantics for cyber-physical systems. In: ICECCS. pp. 123–132. IEEE (2022)
7. Li, R., Zhu, H., Banach, R.: A proof system for cyber-physical systems with shared-variable concurrency. In: ICFEM. Springer (2022)
8. Zhou, C., Wang, J., Ravn, A.P.: A formal description of hybrid systems. In: Hybrid Systems. Lecture Notes in Computer Science, vol. 1066, pp. 511–530. Springer (1995)