Hindawi
Journal of Advanced Transportation
Volume 2020, Article ID 8830752, 24 pages
https://doi.org/10.1155/2020/8830752

Research Article

Multiclock Constraint System Modelling and Verification for Ensuring Cooperative Autonomous Driving Safety

Jinyong Wang[1], **Zhiqiu Huang**[1], **Xiaowei Huang**[2], **Yi Zhu**[3], **and Fei Wang**[1]

[1]*College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, No. 29 General Avenue, Jiang Ning District, Nanjing 211106, China*
[2]*Department of Computer Science, University of Liverpool, Ashton Street, Liverpool L69 3BX, UK*
[3]*School of Computer Science and Technology, Jiangsu Normal University, No. 101 Shanghai Road, Tongshan District, Xuzhou 221116, China*

Correspondence should be addressed to Zhiqiu Huang; zqhuang@nuaa.edu.cn and Xiaowei Huang; xiaowei.huang@liverpool.ac.uk

Received 24 June 2020; Revised 22 October 2020; Accepted 24 October 2020; Published 7 December 2020

Academic Editor: N. N. Sze

CADS (cooperative autonomous driving systems) are software-intensive and safety-critical reactive systems and give great promise to our daily life, but system errors may not be identified in the design stage until the implement stage, and the cost to correct them will be more expensive later than the early stage. For designing trustworthy autonomous software systems, we have to deal with multiclock constraint models. SysML (System Modeling Language) meets increasing adoption in order to carry out system-level modelling and verification against abstract representations, but it suffers from semantic ambiguities in the design of safety-critical autonomous systems. The main objective is to investigate methods for coping with the design and analysis models simultaneously and to achieve semantic consistency based on mathematical foundations and formal model transformation. In this paper, we propose a method to combine the requirement modelling process with analysis process together for CADS safety and reliability guarantee. Firstly, we extend SysML metamodels and construct SysML profile for the CADS domain that could improve modelling correctness and enhance reusability. An instantiated CADS model has been designed by means of adopting a profile containing different key functional and nonfunctional attributes and behaviors. Secondly, we define formal syntax and semantic notations for modelling elements in the SysML state machine diagram and show transformation rules between the state machine diagram and the CCSL (Clock Constraint Specification Language) model. Semantic preservation is also proved using the bisimulation relation between them for rigorous mapping correctness. Thirdly, a cooperative autonomous overtaking driving case study on the highway scenario is used for illustration, and we use the tool TimeSquare to simulate CCSL specification execution traces at the system design stage.

1. Introduction

It can be anticipated that unmanned intelligent systems are increasing rapidly. They can adapt to hostile or hazardous environment and accomplish some extreme tasks, which are difficult or impossible for humans, such as dangerous conditions, extreme speed action or long-duration flights, and cloudy or inclement weather [1]. In recent years, CADS (cooperative autonomous driving systems) are widely used in the connected traffic situation. This application can improve traffic safety and congestion by exchanging interior- and intervehicle communication data. CADS require frequent interaction with surrounding vehicles and the environment. They are the implementation of reactive and control systems and belong to globally asynchronous and locally synchronous systems. We should endeavor in proposing a modelling method for multiclocked and distributed autonomous systems. So, we can analyze behavior events through doing multiclock system verification to identify design errors and predict unexpected faults.

Model-driven and model verification are main techniques for developing safety-critical cooperative and

autonomous cyber-physical systems. Autonomous systems can decide what to do or when to complete desired missions individually or collaboratively and often continuously interact with the environment. Safety-related properties, i.e., bad or unexpected events, can never happen during software execution time and play an important role in people's life and property. When we develop an autonomous system, most of errors are generated during the design stage. In addition, system design and analysis are separated from each other, so it will be very expensive and time-consuming to cope with system errors until the implementation stage. SysML (System Modeling Language) is a widely used modelling approach to support discrete event system engineering, and CCSL (Clock Constraint Specification Language) is the most commonly used language for multiclock behavioral representation. The development and verification of formal multiclock constraint models require deep knowledge about CCSL. We can take full advantage of the two modelling languages just mentioned and show that behavioral models developed in the SysML state machine diagram can be transformed into equivalent CCSL multiclock models so that it is natural and conducive to apply SysML design generality and CCSL verification capability [2].

The complexity of autonomous software systems results in the bottleneck of rigorous specification and verification. Especially, in the first step of software requirements engineering, natural language expressions are low levels consistency and can lead to ambiguity. Although formal languages provide consistent and precise syntax and semantics, it remains a difficult mission for stakeholders to understand and communicate [3–5]. The most shared way to develop safe autonomous systems consists in model-driven designing approach. Model transformation is the prime technique during the process of model-driven software development. The latest practical solution in model transformation is semantic equivalence proof, which can help users to comprehend complicated software systems. We should give special emphasis about model verification to prove semantic consistency and properties preserved in the target model. There are three kinds of common model verification techniques, i.e., formal model checking, theorem proving, and model simulation. The main limitation of theorem proving is that it is short of complete automation and needs a lot of human interaction. Although formal model checking can run automatically, it suffers from significant defect, for example, state explosion problem [6]. Model simulation is based on symbolic execution which requires the operational semantics defining model behavior in reaction to input stimuli. It is a widespread technique helping to generate certification evidence through verifying assumptions.

However, more and more features, such as concurrency, distribution, heterogeneity, and multicore, have been introduced into autonomous software, which lead to three important challenges in the development of CADS. Firstly, it is important and difficult to mitigate ambiguities between nonformal language requirements and formal specification. Natural or semiformal language statements often result in misinterpretations and may lead to ambiguities [7, 8]. Secondly, designing cooperative autonomous systems is a multidisciplinary problem because we should consider dependability requirements, emergent phenomena, accountability for accidents, and supporting evolution. It is essential to propose a special modelling language profile, which customizes reference metamodel for a particular cooperative autonomous domain [9, 10]. Last but not the least, the most challenging issue is to prove semantic preservation between the source and the target model [11, 12]. Complexity of CCSL semantics makes it hard to prove semantics consistency for model transformation. Rigorous proof for semantics preservation will ensure the correctness and effectiveness of communication and understanding among application domain experts, model designers, and system analysts [2, 13, 14].

For these three challenges, our research focuses on multiclock autonomous system specification and simulation to overcome the problems mentioned. There are three main contributions in this article. Firstly, we provide a cooperative autonomous system modelling SysML profile including a conceptual model and interrelationships. So, we can enjoy the benefits of using SysML for breaking down the complex systems into discrete pieces, which are conducive to communicate with disparate developers on different platforms and enhance the understandability of CADS stakeholders. Secondly, we give the formal syntax and semantic notations for modelling elements in SysML and transform the conducted state machine diagram model into mathematically based formal CCSL model. The benefit of transformation is that we can use different models and software safety and reliability analysis tools without having to repeat modelling processes. After presenting syntax and semantics of the SysML state machine and CCSL, we must prove behavior preservation through the proof of bisimulation relation. Thirdly, in order to obtain certification evidence for complex and safety-critical autonomous systems, we adopt a commonly used simulation method in this work. The simulation tool TimeSquare is specifically designed to verify and validate multiclock constraint CCSL models, and we make full use of it to generate execution simulation results to analyze corresponding multiclock constrain autonomous systems.

The structure of this paper is as follows: Section 1 introduces the background and objectives of this paper. In Section 2, motivations and connections with the related work of multiclock constraint autonomous system modelling and verification are given. In Section 3, we present multiclock constraint systems and describe relevant state-of-the-art concepts used for systems' specification and verification, such as model-driven method, CCSL, simulation tool TimeSquare dedicated to CCSL analysis, and motivating scenario. We present the state machine model for CADS, semantic mapping rules from the state machine to the formal CCSL model, and bisimulation proof for behavior equivalence in Section 4. In Section 5, a case study, autonomous overtaking in the highway scenario, demonstrates effectiveness of the proposed method. In the last part of this paper, Section 6 discusses conclusions by summarizing contributions and points out future work.

2. Motivation and Related Work

With the rise and development of advanced driver assistance systems, from almost pure human control to fully autonomous decision with minimal human interaction, they are now being deployed in safety- and mission-critical scenarios. In safety-critical systems, the massive use of software is increasingly improving our daily life, and autonomous vehicle manufacturers and suppliers already design and ensure safety-critical software. Safety is one of the major nonfunctional properties, and tremendous expectations and increasing deployments lead us towards using formal specification and formal verification to obtain evidence for cooperative and autonomous systems [15]. In this section, we describe motivation and short review research related to modelling and verification for multiclock autonomous systems.

2.1. System Modelling and Model Transformation. Autonomous driving system is a kind of multiclock constraint system, so we should provide rigorous modelling methods to ensure safety and correctness. Firstly, we discuss relevant work involving system modelling and model transformation. Bochmann et al. [16] presented the controller synthesis and compositional verification for multilane traffic maneuvers. Arcile et al. [17] proposed a framework using timed automata and model checker UPPAAL (developed by Uppsala University and Aalborg University) to verify safety and robustness properties. Kamali et al. [18] constructed a spatial controller using hybrid agent architecture to model autonomous lane-change maneuvers for real-time and spatial properties. Webster et al. [19] introduced an approach combining formal verification and flight simulation for autonomous unmanned aircraft. Akhtar and Missen [20] demonstrated a method to construct a stepwise refinement multiagent model using process algebra from abstract to concrete concepts incrementally and showed how to ensure the safety and liveness properties for concurrent and interacting cooperative autonomous systems. The work in [21] introduced an abstract model for autonomous urban traffic scenarios using multilane spatial logic and showed that controllers modelled by extended timed automata can be proved safe using spatial reasoning at intersections. The aforementioned papers mainly use formal modelling methods directly at the beginning of requirement engineering. However, semiformal modelling can bridge the gap between natural requirements and formal models.

Now, we will briefly discuss the relationships between our work and existing work about modelling of cooperative autonomous systems. Bernardi et al. [22] extended the UML profile, which can specialize for the railway domain and then used model transformation to generate repairable fault tree and Bayesian network models. Kapos et al. [23] explored a declarative approach and converted query/view/transformation-relation SysML model to executable simulation models, fully conforming model-driven architecture concepts. Caltais et al. [24] wanted to link two worlds of modelling and formal verification through providing transformation rules from SysML to Promela and prove the correctness based on ATL- (Atlas Transformation Language-) based transformation. Dickerson et al. [25] proposed a transformation metamodel between the UML activity model and fault tree model for end-to-end safe development process. Alshboul and Petriu [26] proposed an approach to automatically transform SysML into fault tree implemented in ETL (Epsilon Transformation Language), which can realize the purpose to integrate safety analysis within the system development process. Dias et al. [27] translated software architecture description language based on SysML and specialized profiles to process calculus CSP (communicating sequential process), and the translation can guarantee safety and liveness properties. The aforementioned works show the latest trend of model-based system development modelling and SysML model transformation for software-intensive systems. However, there is still lack of research on model transformation methods for multiclock systems, and we should provide cooperative autonomous system modelling profile and focus on transformation mapping rules for multiclock constraint time-critical systems, which is the first motivation of our work.

2.2. Multiclock System Specification and Semantic Equivalence. After the natural language requirement model is conducted, developers need to verify correctness and confirm whether it meets the customers' expectations. In this section, we review existing works about multiclock specification and semantic equivalence.

Multiclock timing behavior is critical for real-time cyber-physical systems. In order to analyze them early in the design stage, Goknil et al. [28] proposed a method that makes model transformation twice for formal model checking using model checker UPPAAL. The work in [29, 30] presented an approach mapping CCSL to timed input/output automata to verify safety-critical properties. The work in [31] proposed a method to combine synchronous specification language Esterel to perform validation through observers executing CCSL specification. [32–34] extended CCSL with the stochastic semantics and translated it into UPPAAL models to perform formal verification for verifying stochastic and dynamic behaviors. The work in [35] translated probabilistic CCSL into proof objective models for supporting multiclock probabilistic analysis. Chen et al. [36] translated multiclock requirement modelling with CCSL into NuSMV to verify its consistency.

The mentioned works mainly use model transformation for the multiclock CCSL model, and part of them gives the semantic mapping rules, but most of them do not provide rigorous proof for behavior equivalence and model bisimulation. In this paragraph, we list several works considering behavior conservation through model bisimulation. Lam and Padget [37] defined syntax and semantics of UML statechart diagrams and π-calculus, translated statechart diagrams into -calculus, and adopted open bisimulation to check equivalence. Tolbi et al. [38] translated EHPNs (elementary hybrid Petri nets) into the hybrid automata model in terms of timed transition systems and gave behavior preservation proof in the form of timed bisimulation. Bodeveix et al. [39] proved the correctness of transformation

AADL (Architecture and Analysis Design Language) to the target FIACRE language. Baouya et al. [40] introduced a novel verification framework translating the SysML activity diagram into the probabilistic model checker and proved the correctness and soundness of transformation. The work in [41] proposed a method to model stochastic processes with continuous states and proved the behavior equivalence based on the structural operational semantic rule. Bonchi et al. [42] introduced an enhancement up-to bisimulation technique for proving equivalence of open terms. Hülsbusch et al. [43] adopted two bisimularity preservation proof techniques and discussed the proof scalability issue. The work in [44] discussed the bisimulation approach to verify the semantic equivalence.

To sum up, model transformation is an indispensable step of the model-driven development method for safety-critical software, and it is necessary to prove semantic preservation and equivalence between the source model and the terminate model. As for multiclock constraint systems, they are short of behavior reservation proof and simulation at the design stage, which is the second motivation of this work.

2.3. Model Verification for Multiclock Systems. According to the model-driven development approach, after model specification and model transformation, we should focus on model verification before system delivery. For three commonly used model verification methods, because theorem proving needs a lot of human interaction and formal model checking encounters the state explosion problem [45], in this paper, we adopt the model simulation method to perform model verification for multiclock constraint systems.

We have carefully searched for previous work with relation to this study. Do et al. [46] provided a survey on simulation models to verify connected and automated intelligent vehicles and showed that simulation-based analysis guides performance measures. Related works in [47, 48] provided an approach for conjoint simulation to enjoy the benefit of understanding multiclock systems' synchronization at the early design model. Mallet and De Simone [49] proposed two different techniques to perform state-based specification and conduct causal and temporal analysis in the simulation engine. Morando et al. [50] adopted a simulation-based approach using the traffic microsimulator surrogate safety assessment model and suggested that autonomous vehicles can improve traffic safety significantly. Shen et al. [51] proposed an early potential warning system to guide driving behaviors and verified reliability by means of cosimulation. Luo et al. [52] presented an approach to enhance traffic safety through rear-end collision models and numerical simulation. The work in [53] proposed a predictive control model for multivehicle lane-change cooperative maneuver to improve safety of intelligent traffic.

In conclusion, as for multiclock constraint and safety-critical autonomous systems, software concepts, assumptions, and terminologies may be inconsistent because of the limitation of real data. The emergent autonomous systems are composed of multiple clocks, events, and entities which benefit from agent-based modelling and simulations [54], and simulation results provide guidance to identify and avoid potential deadlocks, errors, and hazards as early as possible in the design phase, which is another motivation for our work.

3. State of the Art for Multiclock Systems' Specification and Verification

In this section, we mainly focus on relevant developing methods and technologies for autonomous system design and analysis.

3.1. Model-Driven Development and SysML/MARTE. MDD (model-driven development) is an approach for designing complex software systems, especially cyber-physical systems and autonomous software systems. According to this development method, lower-level systems' artifacts, such as source code, are derived from high-level abstract modelling artifacts. System engineers could not only reduce time consumption through reusability and modularity for researchers and practitioners but also enhance safety and security of the whole system. MDD is based on model-driven architecture and dedicated to innovative system development.

The basic idea of MDD is that everything is a model, and software developing process is driven by system modelling behavior. A model is the abstraction and representation of reality for a given purpose. Although it cannot represent all aspects of the real world, it allows us to deal with problems through avoiding the complexity and improving reusability. These different models stand for different abstract levels for system development. Model-driven developing process is presented in Figure 1. At the beginning of MDD, system engineers perform requirement analysis according to systems' requirements in the form of natural language. Next, they construct a transformation requirement model by sorting out the basic concepts and relationships among all system specifications in the process of the CIM (Computational Independent Model). Then, they transform the CIM into the transformation design model belonging to the PIM (Platform-Independent Model) through M2M (Model-to-Model) transformation, which can transform the source model into another object model [55, 56]. PIM considers logical data that can be abstracted nothing to do with any application platforms. We obtain a platform-specific model (PSM) by adding concrete and related platform information in the PIM. Through the subsequent M2M transformation, PIM can be transformed into PSM. Finally, source code (or other code-related software artifacts) is obtained through Model-to-Text transformation. In the process of PIM-to-PSM transformation, metamodel and profile support different abstraction model construction, and transformation metamodel defines the abstract syntax and static semantics of the corresponding metalanguage. The profile defining the concrete syntax of the language extends a reference metamodel to adapt for specific platform, particular domain, or a software development method. The transformation model

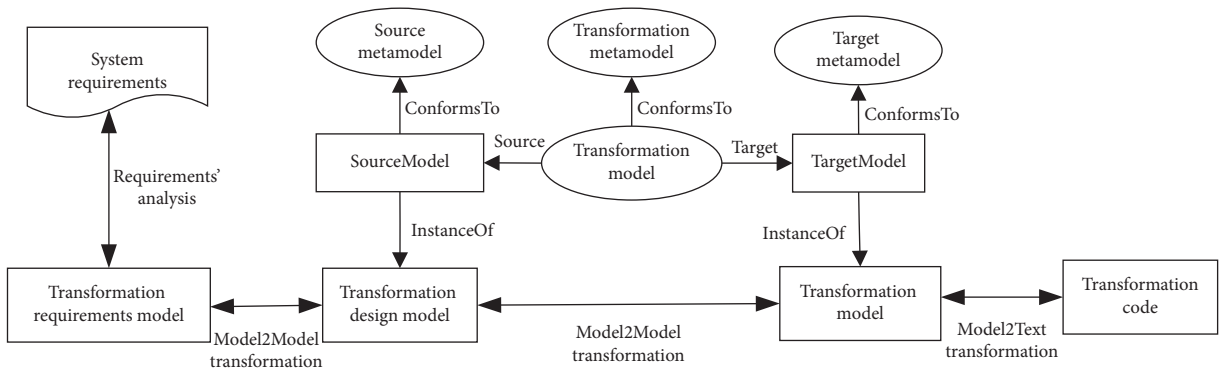

Figure 1: Modeling process and transformations in MDD.

consisting of transformation rules conforms to the transformation metamodel and maps constructors from the source model to constructors of the target model, which are in conformance with the source metamodel and target metamodel, respectively. When transformation is executed, transformation function receives the source model instantiated from the PIM as the input and generates the target model instantiated from the PSM as the output.

There are many languages supporting MDD, such as UML (Unified Modeling Language), SysML, CCSL, AltaRica, and AADL. SysML is a graphical modelling language based on general purpose, and it supports modelling hardware and software codesign, coping with high complexity and avoiding traditionally time-consuming development [57]. The purpose of SysML is to describe, analyze, design, and verify complex systems. CADS are a safety-critical and complex system, so it is necessary to perform formal analysis at the stage of design phase for ensuring safety of autonomous systems. One step of model transformation is needed to transform the informal or semiformal model into the formal model for rigorous and comprehensive analysis. A model transformation is similar to a program responsible for transforming one representation into another. Main M2M transformation approaches include relational/declarative, imperative/operational, graph-based, and hybrid. This work mainly pays attention to relational/declarative approaches, and transformation specification languages focus on the mapping relationship between input elements and their corresponding output elements. So, we should define the mapping relationship using mathematical specification predicates and input-output constraints [58].

MARTE system (Modeling and Analysis of Real-Time and Embedded system) combines the latest research results of industry and academia in the embedded field, and it is an open and extensible modelling specification through stereotype, tagged-value, and constraint to extend SysML's modelling capability. Its architecture can be classified into two categories dealing with quantitative and qualitative aspects, respectively, and it is composed of some subprofiles, as is shown in Figure 2. It mainly contains four subextension packs, including foundations, design model, analysis model, and annexes, in the MARTE library. Time package including in MARTE foundations is the main profile for modelling real-time behaviors, which defines time structure, time access, time usage, and other necessary time modelling elements and methods. Time access defines concepts and specifications of representing time structures among these time packages related to this work, such as clock, clock type, and current time. The "foundations" package is made up of the following profiles: (1), core elements which support to model the systems' operational modes; (2), nfp profile which provides modelling constraints for specifying, defining, and applying nonfunctional information to a SysML model; and (3), time profile which contains concepts of time structure, time access, and time usage, which are the main modelling and verifying modes for embedded real-time systems. When CCSL starts to appear, it is a textual specification and companion language complementary to the time profile allowing us to describe the clock constraints in MARTE annexes, and now, CCSL has been fully developed as an independent specification language for logical clock and chronometric clock. Then, we introduce CCSL relevant definitions and concepts and give primary syntax and semantics of the CCSL.

3.2. Syntax and Semantics of the CCSL. Clocks in the CCSL can be seen as events, and their instants stand for event occurrences. They can be logical or physical clocks and dense or discrete clocks [48]. In this paper, we mainly consider discrete clocks. Clocks with the arrival of logical sequence describe behaviors of systems instead of arrival of physical data. In order to describe multiclock systems more conveniently, CCSL allows each clock to be defined independently and only needs to describe existing system behavior and logical restriction relationship between clocks, without assuming total reference clock of a system, which is more conducive to the flexibility of system description. CCSL can specify constraints and evolutions among clocks, which are presented in the form of clock relations or clock expressions. The clock relation may be synchronous or asynchronous, and we can use clock expression to define new clocks based on other existing clocks. The core concept of multiclock constrained language CCSL is that a logical clock consists of a series of moments (instants), a moment corresponding to a tick of the clock. Now, we elaborate on preliminary definitions.

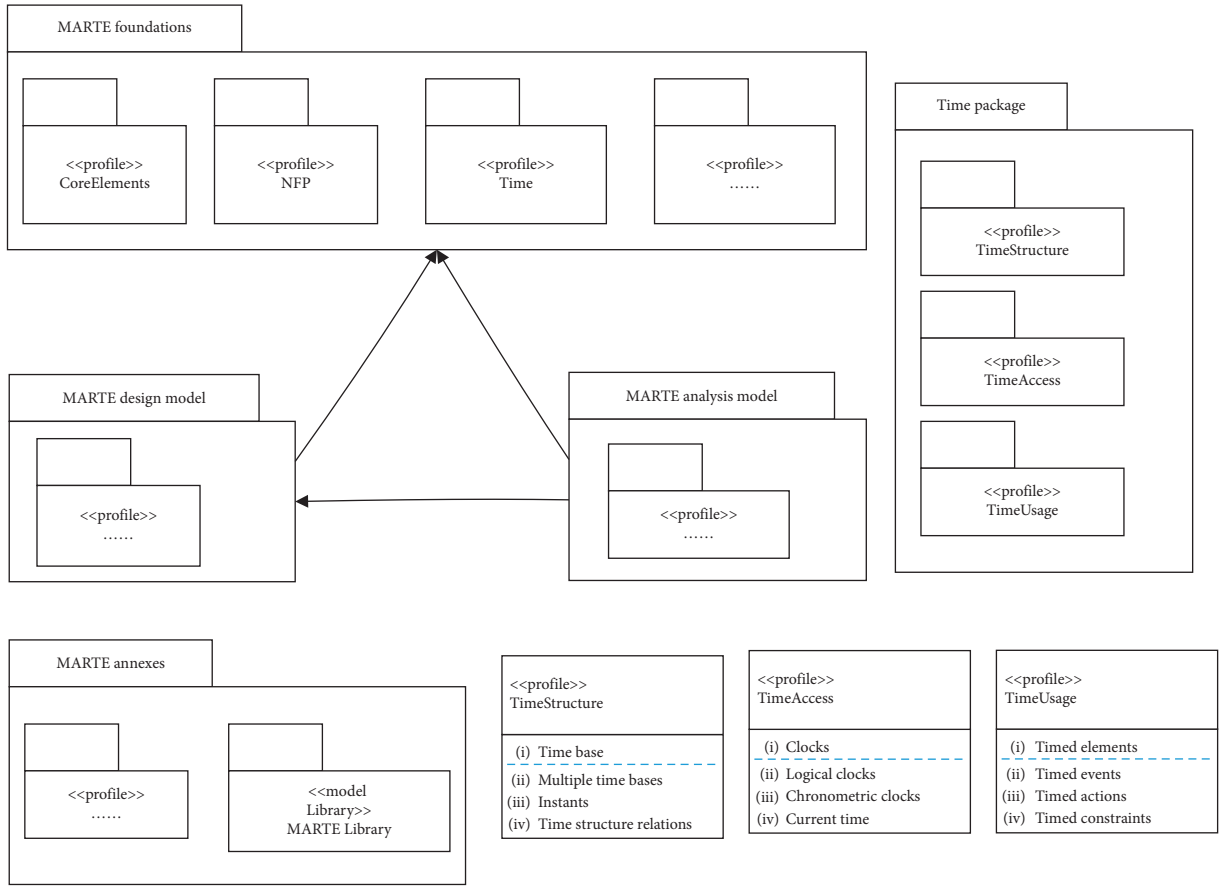

Figure 2: View of MARTE architecture and time package.

At the beginning, we give the definition of the logical clock and multiclock constraint system referring to synchronous modelling language SIGNAL [59]. Logical clock, introduced by L. Lamport, is the central concept in CCSL, which abstracts physical time to partial order of SysML events. Clock constraint system can be divided into two categories, Monoclock system and multiclock system.

Definition 1 (logical clock). In CCSL, a logical clock is a quintuple notation, $\langle \text{Ins}, <, \text{Lab}, \lambda_{\text{cl}}, U \rangle$, which represents a set of ordered instants. Formally, a clock $c = \langle \text{Ins}, <, \text{Lab}, \lambda_{\text{cl}}, U \rangle$, where Ins is a sequential set of instants, $<$ is a quasi-order relation on Ins, Lab is a set of labels, λ_{cl}: Ins $\longrightarrow$ Lab is a labeling function, and U stands for logical clock units and is often called tick abstractly. For any discrete logical clock, $c(i)$ denotes i^{th} instant in Ins_c, and for each $c(i) \in \{\text{tick}, \text{idle}\}$, Ins_c is the set of occurrences or ticks for clock c. Now, we give relevant definitions for initial and terminal instants, respectively.

(1) $i \in \text{Ins}_c$ is the initial instant of clock c if and only if $\forall j \in \text{Ins}_c/\{i\},\ (i, j) \in <_c$

(2) $i \in \text{Ins}_c$ is the terminal instant of clock c if and only if $\forall j \in \text{Ins}_c/\{i\},\ (j, i) \in <_c$

For any instant $i \in \text{Ins}_c$, $^\circ i$ and i° stand for the predecessor and successor of i in Ins_c, respectively. If i is neither the initial instant nor terminal instant, $\exists m \in \mathbb{N}$, $\mathbb{N}$ is the set of natural numbers, which makes $c(m) = i$, then $c(m-1) = {}^\circ i, c(m+1) = i^\circ$; and if $i \in \text{Ins}_c$ is the initial instant, then there is only successor instant, and there is no predecessor for i; if $i \in \text{Ins}_c$ is the terminal instant, then there is only predecessor instant, and there is no successor for i. Now, we give an example for the logical clock, and the graphical representation of clock April is shown in Figure 3.

clock April = $(\{1, 2, \ldots, 29, 30\},$
$<, \{2020.4.1, 2020.4.2, \ldots, 2020.4.29, 2020.4.30\}, \lambda, \text{day})$.

(1) $<: i < j, i \in \{1, 2, \ldots, 29\} \wedge j \in \{2, 3, \ldots, 30\} \wedge j = i + 1$.

(2) λ_{cl}: Ins $\longrightarrow D$, $\forall i \in I, \lambda_{\text{cl}}(i) = 2020.4.i$.

Definition 2. Monoclock system and multiclock system

A monoclock system is a system in that all activations of subsystems (also called subclocks) are controlled by a single and global master clock; a multiclock system is a system composed of several functionally independent systems, in which each system holds its own activation clock, and there is no master clock existing in the whole system.

In a monoclock model, each component's clock would have a strict dependency relationship to the system's global master clock, which would result in tight coupling between subsystem clocks and the global clock. Once one

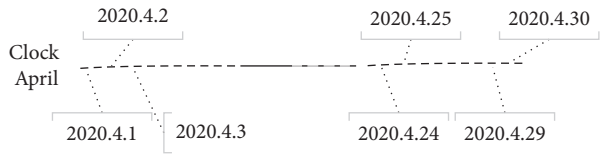

Figure 3: Graphical representation of clock April.

component's clock frequency changes, the global clock and other subclocks also need to be adjusted accordingly. Multiclock systems are globally asynchronous and locally synchronous distribute systems, which are called the formal modelling framework polychrony [60, 61]. Components contained in the multiclock system obey to multiple clock rates, it has no global clock, and each component works according to its own clock, which is loosely coupled with other clocks, and clock synchronization takes place only between components that interact. A clock in the timed automata model captures physical real-time value properties, which will be elapsed as transition execution, and all clocks evolve according to the same rate [62]. Therefore, a multiclock model is suitable for modelling distributed or highly parallel systems. In monoclock and multiclock systems, the clock mainly represents logical time properties, and graphical representation for them is shown in Figure 4.

Definition 3 (clock schedule). A schedule is a function Sched: $\mathbb{N} \longrightarrow 2^c$. Given an execution step $i \in \mathbb{N}$ and clock schedule $\sigma \in$ Sched, σ is an infinite or finite sequence $\sigma(0)\sigma(1), \ldots, \sigma(i), \ldots, i \in \mathbb{N}$, C is a finite set of clocks, and each $\sigma(i) \in 2^c$ denotes a set of clocks that tick at step i.

$$\sigma(i) = \begin{cases} \varnothing, & i = 0 \\ \{c | c \in C \wedge c(i) = \text{tick}\}, & i \geq 1 \end{cases}.$$

Definition 4 (clock history). Given any schedule σ, the clock history H_σ is a function H_σ: $C \times \mathbb{N}^+ \longrightarrow \mathbb{N}$, which keeps inductive record of the number of ticks for each clock up to current time. It is defined recursively as

$$\begin{aligned} H_\sigma(c,i) &= \left|\{j | j \in \mathbb{N}^+, j \leq i, c \in \sigma(j)\}\right| \\ &= \begin{cases} H_\sigma(c,0) = 0, & i = 0, \\ H_\sigma(c,i+1) = H_\sigma(c,i), & \forall i \in \mathbb{N}, c \notin \sigma(i), \\ H_\sigma(c,i+1) = H_\sigma(c,i) + 1, & \forall i \in \mathbb{N}, c \in \sigma(i). \end{cases} \end{aligned} \quad (1)$$

Definition 5 (CCSL specification). A CCSL specification is an ordered pair relation Spec = $\langle C, \text{Cons} \rangle$, where C is the set of logical clocks in Definition 3, and Cons is a finite set of clock constraints, which include clock relations and clock expressions. Clock relation consists of the connection of the logical clock and the binary operator and can be synchronous or asynchronous. The basic clock relation syntax is defined as follows.

Rel::=$c_1 \prec c_2 | c_1 \# c_2 | c_1 \subseteq c_2 | c_1 \equiv c_2 | c_1 \sim c_2 | c_1 \preccurlyeq c_2$, where c_1 and c_2 are any clocks belonging to the set C. The semantics of clock relations is shown in Table 1.

The semantic interpretations of clock relations are explained informally as follows: "precedence" means that each instant of c_2 can only occur after the corresponding instant of clock c_1; "exclusion" means that any two clocks c_1, c_2 cannot happen at the same instant; "subclock" means that c_1 is a subclock of its superclock c_2, and every instant of clock c_1 coincides synchronously with one of the instants of clock c_2; when clock c_1 and clock c_2 satisfy $c_1 \subseteq c_2 \wedge c_1 \supseteq c_2$, two clocks are said to be synchronous and can be denoted as $c_1 \equiv c_2$; "causality" means that when an event causes another one, the result event cannot occur if the cause event does not happen, so every clock c_1 has to precede clock c_2 in the same instant in schedule σ; and "alternation" relation $\sim$ is derived from the precedence relation $\prec$ shown in Table 1. $\sim$ is a form of bounded $\prec$, and clock c_1 alternates with c_2 which is denoted as $c_1 \sim c_2$ or $c_1 \prec_1 c_2$, and an instant of $c1$ precedes the same instant of c_2 which in turn precedes the next instant of c_1. A possible schedule of clock relations is shown in Figure 5. In the result of simulation of TimeSquare presence, there are two kinds of links between two different ticks: red link, which stands for coincidences (strongly synchronous) and blue arrows, which stand for causalities (weakly synchronous).

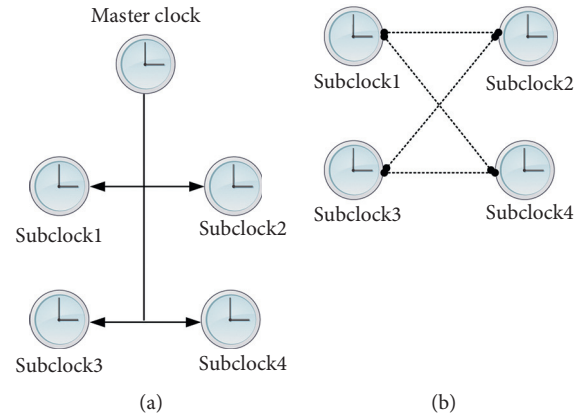

Figure 4: (a) Monoclock system and (b) multiclock system.

Clock constraints can either be clock relations or clock expressions. Every clock expression is a set of clock definitions, and we can use an expression to define a new clock based on a set of parameters. The basic clock expression syntax is defined as follows.

Exp::=$c_1 + c_2 | c_1 * c_2 | c_1 \blacktriangleright \quad c_2 | c_1 \quad \downarrow \quad c_2 | c_1 \vee c_2 | c_1 \wedge c_2 | c_1^{\wedge} n$, where c_1 and c_2 are any clocks belonging to the set C. The semantics of clock expressions is shown in Table 2.

The semantic interpretations of clock expressions shown formally in Table 2 are explained in natural language informally as follows: the result of the "Union" is a clock c_0 which ticks whenever either c_1 or c_2 ticks; "Intersection" expression defines a clock c_0 which ticks whenever both c_1 and c_2 tick at the same instant; expression "DelayFor" defines a clock c_0 which is synchronous with part of the delay clock c_2, and every tick of the base clock c_1 starts up the counter d which will be decreased with every tick of the delay clock c_2; when the counter d reaches 1, the clock c_0 will occur along with the delay clock c_2 at the same time; "PeriodicOn" defines a clock c_0 which ticks periodically every p^{th} tick of another base clock c_1; "Sample" produces the fastest clock c_0

Table 1: The semantics of clock relations.

Relation name	Notation	Kind of relation	Semantics of relations
Precedence	$c_1 \prec c_2$	Asynchronous	$\forall i \in \mathbb{N}^+,\ (H_\sigma(c_1,i) = 0 \wedge H_\sigma(c_2,i) = 0) \vee (H_\sigma(c_1,i) > H_\sigma(c_2,i))$
Exclusion	$c_1 \# c_2$	Asynchronous	$\forall i \in \mathbb{N}^+,\ c_1 \notin \sigma(i) \vee c_2 \notin \sigma(i)$
Subclock	$c_1 \subseteq c_2$	Synchronous	$\forall i \in \mathbb{N}^+,\ c_1 \in \sigma(i) \longrightarrow c_2 \in \sigma(i)$
Coincidence	$c_1 \equiv c_2$	Synchronous	$\forall i \in \mathbb{N}^+,\ c_1 \in \sigma(i) = c_2 \in \sigma(i)$
Causality	$c_1 \preccurlyeq c_2$	Asynchronous	$\forall i \in \mathbb{N}^+,\ H_\sigma(c_1,i) \geq H_\sigma(c_2,i)$
Alternation	$c_1 \sim c_2$	Asynchronous	$\forall i \in \mathbb{N}, c_1(i) \prec c_2(i) \wedge c_2(i) \prec c_1(i+1)$

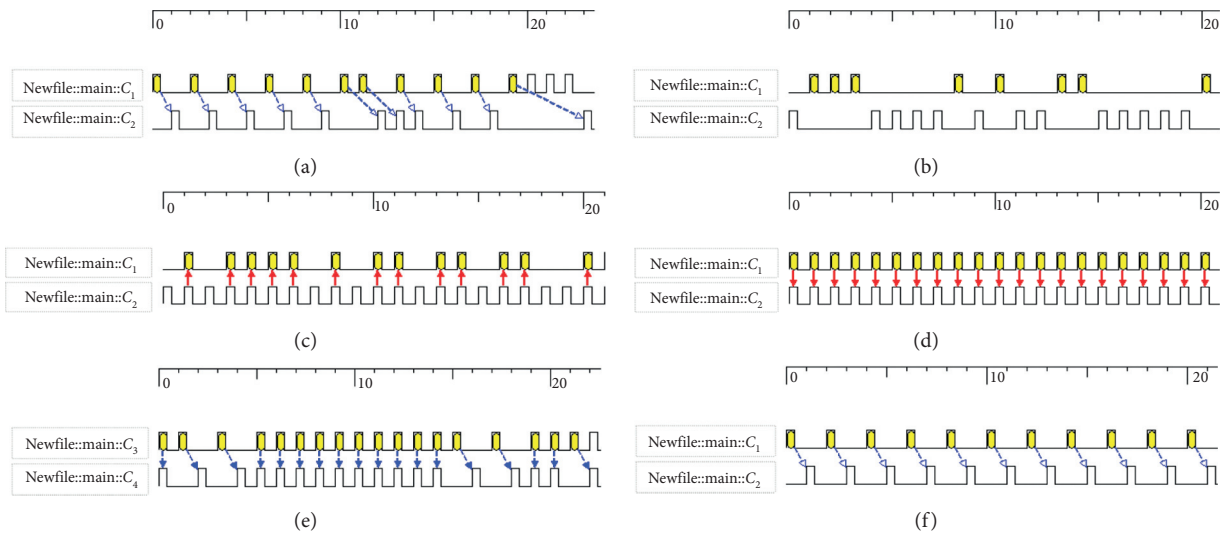

Figure 5: A possible schedule of clock relations: (a) $c_1 \prec c_2$, (b) $c_1 \# c_2$, (c) $c_1 \subseteq c_2$, (d) $c_1 \equiv c_2$, (e) $c_1 \preccurlyeq c_2$, and (f) $c_1 \sim c_2$.

slower than c_1 that is a subclock of base clock c_2, and it is synchronous with the base clock c_2; it is easy to show that $c_0 \subseteq c_2 \wedge c_1 \preccurlyeq c_0$; "StrictSample" defines a clock c_0 that satisfies the formula $c_0 \subseteq c_2 \wedge c_1 \prec c_0$; "Supremum (Infimum)" defines the slowest (fastest) clock c_0 which is faster (slower) than both clocks c_1 and c_2; "Preemption (UpTo)" produces a resulting clock c_0 ticking in coincidence with the clock c_1 and dying as soon as the clock c_2 starts to tick; and "Wait" defines a clock c_0 which will tick only once n^{th} for the corresponding base clock c_1, and then the resulting clock dies forever. A possible schedule of clock expressions shown in Table 2 is presented in Figure 6.

3.3. Model Verification and TimeSquare. After the system model is built according to the model-driven development method, one of real problems is that we need to perform model validation and verification to ensure correctness and safety of the whole system. There are various techniques used to perform model validation and verification, such as formal model checking, theorem proving, abstract interpretation, simulation, and emulation. Among these model verification techniques, simulation models are comparatively flexible to find unexpected problems at the system design phase and can be modified to accommodate changing environment to a real situation. In this paper, we focus on model simulation after constructing a conceptual model. TimeSquare is an Eclipse and model-driven software environment to cope with the MARTE time model and CCSL. This Eclipse plugin can be used to perform interactive clock-related specification, check clock constraints, and give a waveform solution [47].

CCSL has rigorously formal semantics making CCSL multiclock constraint specification executable in TimeSquare. According to Definition 4 (clock history), we call $H_\sigma(c,i)$ a configuration of clock c at instant i. So, a configuration is a set of enabled clocks (ticking) at a given step. At each step, the result of TimeSquare consists of a Boolean solution and a set of all valid configurations. If the polychromous clock constraint is deterministic, it will produce only one valid configuration; if it is nondeterministic, one of the possible configurations will be chosen in accordance with the solution policy, which is offered by TimeSquare among several simulation policies. When the clock specification is correct, TimeSquare will generate a valid sequence of steps, and this means that there exists a trace to satisfy clock constraints. If deadlocks are found by the solver after a finite sequence of steps in the waveforms, inconsistent specification should exist in clock constraints. The resulting traces are shown in the VCD (Value Change Dump) format, which is an IEEE standard format used by logic simulation tools. The relations between clocks provided in the TimeSquare tool are partial-order traces, and this illustrates that clocks (events) satisfy constraint specifications during simulation, early design validation, and verification.

Table 2: The semantics of clock expressions.

Clock expression name	Notation	Kind of expression	Semantics of expressions
Union	$c_0 \triangleq c_1 + c_2$	Synchronous	$\forall i \in \mathbb{N}, c_0 \in \sigma(i) \Longleftrightarrow (c_1 \in \sigma(i) \vee c_2 \in \sigma(i))$
Intersection	$c_0 \triangleq c_1 * c_2$	Synchronous	$\forall i \in \mathbb{N}, c_0 \in \sigma(i) \Longleftrightarrow (c_1 \in \sigma(i) \wedge c_2 \in \sigma(i))$
DelayFor	$c_0 \triangleq c_1$	Synchronous or asynchronous	$\exists i \in \mathbb{N}, c_0 \in \sigma(i) \Longleftrightarrow c_2 \in \sigma(i) \wedge \exists j \leq i, c_1 \in \sigma(j) \wedge H_\sigma(c_2, i) - H_\sigma(c_1, j) = d$
PeriodicOn	$c_0 \triangleq c_1 \propto p$	Synchronous or asynchronous	$\forall i \in \mathbb{N}, c_0 \in \sigma(i) \Longleftrightarrow H_\sigma(c_1, i) = p * H_\sigma(c_0, i) \wedge c_1 \in \sigma(i)$
Sample	$c_0 \triangleq c_1 \triangleright c_2$	Synchronous or asynchronous	$\forall i \in \mathbb{N}, c_0 \in \sigma(i) \Longleftrightarrow (c_2 \in \sigma(i) \wedge (\exists 0 < j \leq i)(\forall j \leq k < i) \cdot (c_1 \in \sigma(j) \wedge c_2 \notin \sigma(k)))$
StrictSample	$c_0 \triangleq c_1 \blacktriangleright c_2$	Synchronous or asynchronous	$\forall i \in \mathbb{N}, c_0 \in \sigma(i) \Longleftrightarrow (c_2 \in \sigma(i) \wedge (\exists 0 < j < i)(\forall j \leq k < i) \cdot (c_1 \in \sigma(j) \wedge c_2 \notin \sigma(k)))$
Supremum	$c_0 \triangleq c_1 \vee c_2$	Synchronous or asynchronous	$\forall i \in \mathbb{N}, H_\sigma(c_0, i) = \min(H_\sigma(c_1, i), H_\sigma(c_2, i))$
Infimum	$c_0 \triangleq c_1 \wedge c_2$	Synchronous or asynchronous	$\forall i \in \mathbb{N}, H_\sigma(c_0, i) = \max(H_\sigma(c_1, i), H_\sigma(c_2, i))$
Preemption (UpTo)	$c_0 \triangleq c_1 \downarrow c_2$	Synchronous	$\forall i \in \mathbb{N}, c_0 \in \sigma(i) \Longleftrightarrow (c_1 \in \sigma(i) \wedge \forall 0 < j \leq i, c_2 \notin \sigma(j))$
Wait	$c_0 \triangleq c_1 \hat{} n$	Synchronous	$c_0 \in \sigma(n) \wedge \forall i \in \mathbb{N}, i > n, c_1 \notin \sigma(i)$

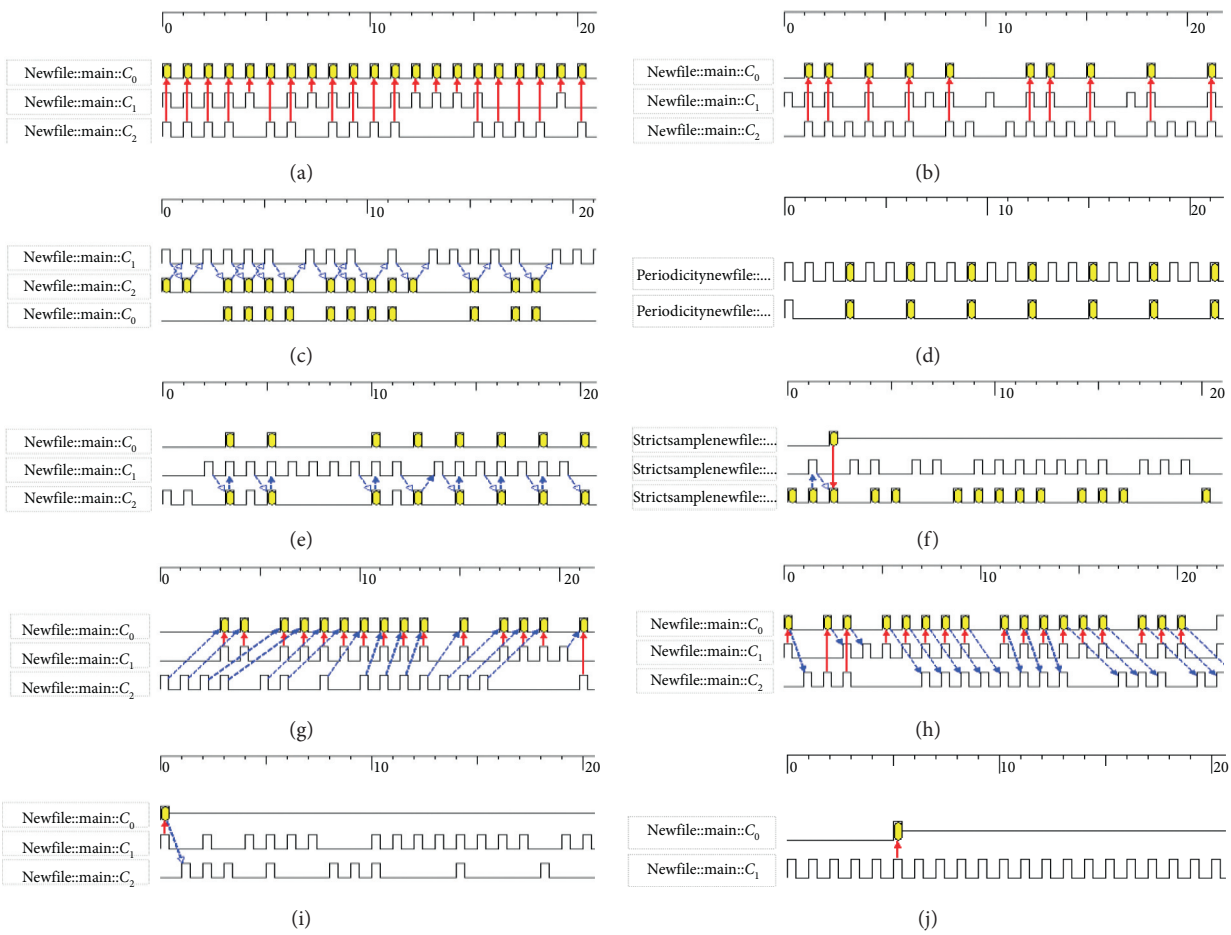

Figure 6: A possible schedule of base clock expressions: (a) $c_0 \triangleq c_1 + c_2$, (b) $c_0 \triangleq c_1 * c_2$, (c) $c_0 \triangleq c_1$, (d) $c_0 \triangleq c_1 \propto p, p = 3$, (e) $c_0 \triangleq c_1 \triangleright c_2$, (f) $c_0 \triangleq c_1 \blacktriangleright c_2$, (g) $c_0 \triangleq c_1 \vee c_2$, (h) $c_0 \triangleq c_1 \wedge c_2$, (i) $c_0 \triangleq c_1 \downarrow c_2$, and (j) $c_0 \triangleq c_1 \wedge n, n = 6$.

3.4. Motivating Scenario: The Architecture of CADS. An autonomous car can be considered as a computer-aided and computer-controlled vehicle which monitors, percepts surroundings, guides itself, makes decisions, and is fully controlled and operated without any human interactions [63]. In fact, CADS are a kind of intelligent systems, such as reactive systems, self-adaptive systems, cyber-physical systems, and ubiquitous intelligent operating systems. These types of systems have a common characteristic that requires constant interaction with the environment continuously, but

the main difference and advantage of the autonomous system are that it can "see" or percept the environment, "hear" or communicate with surroundings, and react fast enough and control independently without humans or other individual intervention. The purpose of autonomous driving system emergence is to improve driving safety, reduce environmental pollution, and ease the traffic congestion situation, but if these systems only act autonomously and lack connectivity and cooperativeness, autonomous driving systems may lead to collision, chaos, and collapse in the case of high densities and large-scale situation. Therefore, autonomous driving system must be designed not only to be interconnected but also to be cooperative. In this paper, we only consider connected and cooperative autonomous driving system because this kind of system can negotiate with other infrastructures and vehicles to share common information for safety and driving comfort level.

CADS can communicate with other vehicles and infrastructures to obtain environment information; the system reacts fast enough to produce corresponding outputs. The system must cope with input events and output events, and inputs must precede outputs strictly. Events can be seen as multiclocks, and consumed time of computations and communications can be abstracted as instantaneous. Every computation time can be treated as an instant rather than a time interval; in other words, we can abstract and assume that each calculation time is zero, and meanwhile, the sequence of events (or clocks) must satisfy functional specification. This kind of model is a synchronous model. It is a type of PIM, which does not care about calculation time but only focuses on the sequence of the input event. So, synchronous model for real-time systems only cares about system functional properties and provides a high-level abstract model for system modelling and verification. In order to introduce the process of specification and simulation for CADS, we should present the architecture and basic elements in the autonomous driving system. SysML profile for the CADS architecture model and knowledge model is designed and shown in Figure 7. In the design stage of system development, it is necessary to model structural and behavioral aspects in autonomous systems.

The architecture model contains all blocks and functional operations regarding the user, environment, and autonomous vehicles. Model safety is found on the basis of formal description. The detailed definition is as follows.

Definition 6. The autonomous architecture model (AAM) is a quadruple notation $\mathrm{AAM} ::= (B_A, R_A, E_A, C_A)$:

(1) B_A refers to the finite set of autonomous system blocks and is constructed based on the extended BDD (Block Definition Diagram) stereotype. B_A can be expressed as

$$B_A = \{\text{Perception, Understanding, Planner, Actuator, User, Environment, KnowledgeBase}\}. \quad (2)$$

(2) R_A refers to the finite set of autonomous system relationships, which represent connections between functional blocks. R_A can be expressed as $R_A \subseteq B_A \times B_A$, $R_A = \{\text{monitor, trigger, precedence, stimulus, response, invoke}\}$, which is used to describe the interaction between autonomous software blocks, and its semantic interpretation is shown in Table 3.

(3) E_A refers to the function of autonomous system operation $O_A \subseteq B_A \cdot O_A$ evolution, which stands for each functional block operation type. Different blocks have different functional operations and belong to their own operation subset which is related to the corresponding attributes and will evolve when activities of operation occur. E_A: $B_A \times R_A \longrightarrow B_A$ stands for the state transition; as for any operation $B_A \cdot O_A$, when it occurs and satisfies transition conditions, the system will evolve to the next state. For example, if the vehicle's operations occur in block perception, such as Vehicle-to-Vehicle (V2V), Vehicle-to-Infrastructure (V2I), and other perceive actions through sensor tools (Lidar, Radar, Cameras, VisualSensors, etc.), for perceiving TrafficSnapshot (OccupyLane, ClaimLane, Position, Speed, Acceleration, and SafetyDistance), the operations (Observe, DataProcess, and Conform) will be triggered according to TrafficRule and PrioriInformation.

(4) C_A refers to the finite set of conditions, which defines constraint information in the process of autonomous software execution. As for the <<stereotype>> Knowledge Base, S_A = {Invariant, Precondition, Postcondition}.

4. Proposed Methodology in This Paper

In practice of model-driven software development, it is increasingly clear that indispensable design and analysis processes should be integrated together. Design model using design language should be converted into the analysis model using analysis language in order to perform model verification and analysis on the transformed analysis model for safety. Model transformation technology is used to perform this conversion, and transformation process should preserve behavior semantics of the source design model. In this section, we propose an approach to transform the design model into the analysis model and then verify semantic equivalence for this conversion through the model bisimulation proof. First of all, SMD (state machine diagram) for CADS is introduced in the following section.

4.1. Syntax and Semantics of SMD for CADS. In the previous Section 3.4, we use the SysML BDD structure diagram to obtain static specification for the CADS scenario. BDD depicts classes of the major components, attributes, corresponding operations, and relationship between them. In order to ease comprehension of CADS behavior and analyze simulation execution traces, we should construct SysML dynamic diagrams to specify interactive behavior between the system and the surrounding environment and then perform model transformation for the formal model to verify and validate safety properties. SMD is constructed

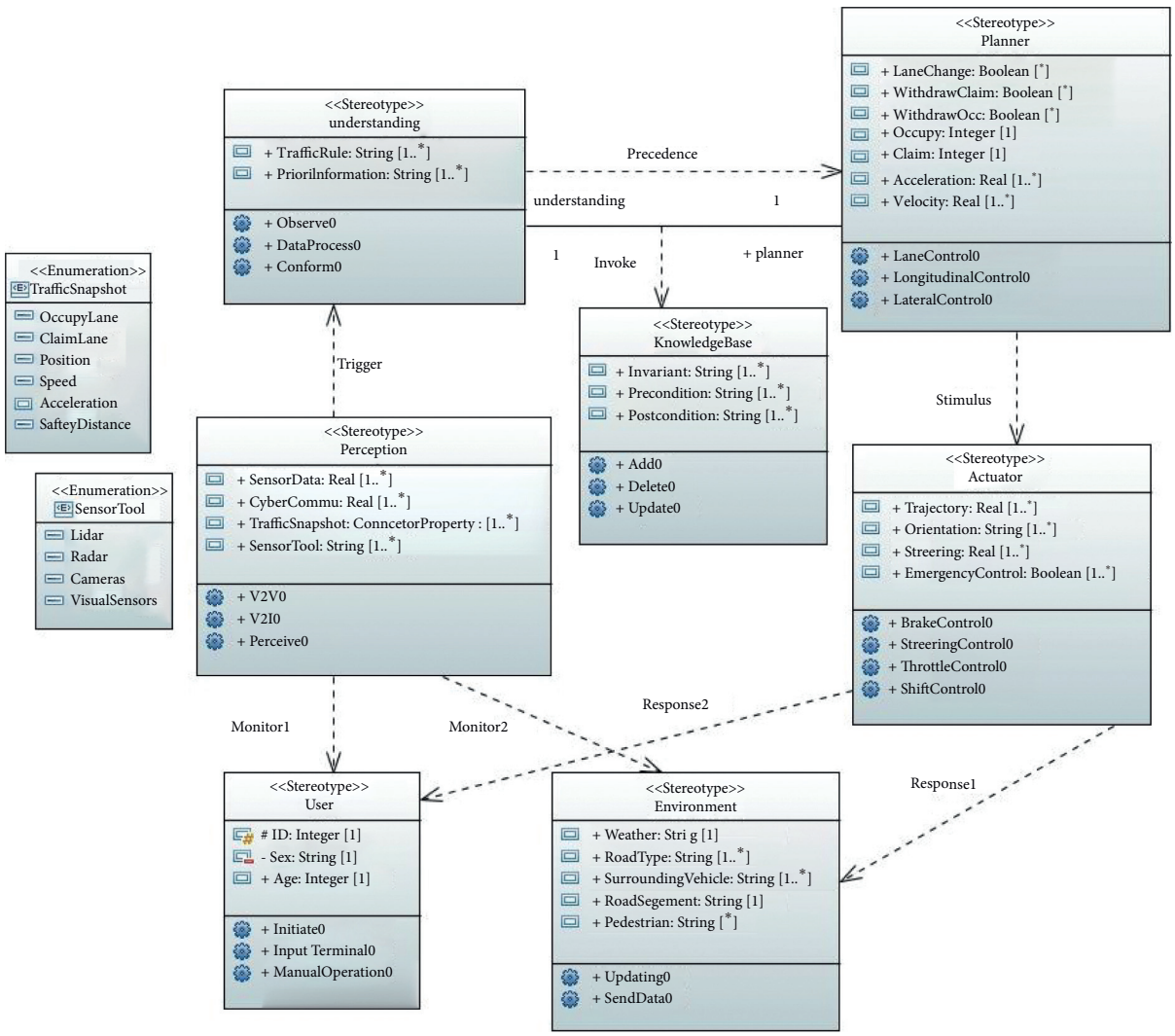

Figure 7: SysML profile for autonomous driving system architecture.

Table 3: Semantic interpretation of relationship in autonomous software.

Relation	Semantics	Interpretation of relation semantics
Monitor	$A \longrightarrow_M B$	Represents the relation of monitor; A will periodically monitor B'sB's operation status and data
Trigger	$A \longrightarrow^C B$	If the condition C in block A is satisfied, the operation in block B will be executed
Precedence	$A; B$	After the operations in block A are completed, operations in block B will be executed successively
Stimulus	$A; B[c_1] \Vert C[c_2]$	As for block A and the relevant conditions c_1 and c_2, if c_1 (c_2) is satisfied, the operations in block B(C) will be executed
Response	$A \longrightarrow_R B$	Block A will adjust the parameters, behaviors, and structures of block B according to the corresponding policies
Invoke	$A \longrightarrow_I B$	Represents the relation of invoke; block B stores the model and data needed for implementation of A

from states and transitions and is very convenient for clearly describing the dynamic behavior transition and state changes in the system. A state is denoted as a rounded rectangle, whereas transitions are arrows from one state to another labelled with three optional parts: a trigger event, a guard condition, and an effect (a sequence of actions) [27]. Now, we present the main features of the state machine for CADS to show system reactive behaviors.

First of all, we can define SMD as this 6-tupleSMD::= $(S, S_0, E, G, \text{Act}, \text{Tr})$.

4.1.1. Vertices and Transitions. A vertex is a node in SMD, which may stand for a state, a final state, or a pseudo-state. A transition refers to a directed labelled edge connecting a source vertex and the corresponding target vertex. A transition may be a compound transition which is composed of multiple transitions connected via fork, junction, or join pseudo-state. Let Tr be a set of transitions; we can use the following notation for Tr: tr: sv $\xrightarrow{e(g)/a}$ tv, and for a transition tr and a set of vertices V, sv $\in V$ and tv $\in V$ are the source and target vertex of the corresponding transition tr;

$e \in E, g \in G, a \in$ Act are the trigger event, guard condition, and effective behaviors associated with tr, respectively. The semantics of transition function tr is that if the trigger event occurs and Boolean guard conditions are satisfied, then the sequence actions would be executed, and the system translates from the current state into the successive state. Especially, trigger events can be classified into external environmental events and internal state events due to autonomous systems' new features. Through this distinction, autonomous systems can percept the external trigger event, such as weather condition, traffic sign, traffic code, road conditions, and surrounding sensors' input data.

4.1.2. Regions. A region is the container of vertices and transitions and indicates orthogonal parts which may be a composite state or a substate machine. Figure 8 contains an orthogonal substate machine region ($s5$: ClaimLane).

4.1.3. States. There are three types of states: type: $S \longrightarrow$ (simple, region, composite). Simple state, e.g., in Figure 8, ($s1$: TurnLeft) is not further refined; region state, e.g., in Figure 8, ($s5$: ClaimLane) is composed of at least two orthogonal states; composite state indicates that the state can be further refined. Each state may have an optional action associated with it: entry, exit, and activity. We can use the type function to avoid unnecessary redundancy and invalid attribute combinations. Let S be a given set of states; S_0 refers to the initial state in SMD and for every state $s \in S$ satisfying the following relation.

$$\text{type}(s) = \begin{cases} \text{simple iff } s.\text{issimple} \wedge s.\text{isregion} \wedge s.\text{iscomposite}, \\ \text{region iff } s.\text{issimple} \wedge s.\text{isregion} \wedge s.\text{iscomposite}, \\ \text{composite iff } s.\text{issimple} \wedge s.\text{isregion} \wedge s.\text{iscomposite}. \end{cases} \tag{3}$$

4.1.4. Pseudo-States. A pseudo-state is a vertex in the state machine connecting multiple transitions into more complex paths. Pseudo-states are extensions of state machine syntax in order to express rich enough behaviors. We can define a pseudo-state as a tuple: ps = $\langle r, f \rangle$; $r \in$ region defines the region to which the pseudo-state belongs; $f \in S$ is an option recording the last active state field and is only used in shallow history or deep history pseudo-state conditions. There are approximately ten kinds of pseudo-states defined in SysML SMD:

$$ps = \begin{pmatrix} \text{initial, exitpoint, choice, join, fork, junction,} \\ \text{terminate, entrypoint, shallowhistory, deephistory.} \end{pmatrix}. \tag{4}$$

Then, we define SOS (structured operational semantics) of SMD for CADS using LTS (labelled transition system). The definition of LTS is shown as follows.

Definition 7. A LTS is a 6-tuple LTS$::=(S, \text{Act}, \longrightarrow, \text{Init}, \text{AP}, L)$ such that

S is a set of states (the state space).

Act is the set of actions.

$\longrightarrow \subseteq S \times \text{Act} \times S$is a relation of the transition. We can denote the transition $(s, \alpha, s') \in \longrightarrow$ as a short notation $s \xrightarrow{\alpha} s'$.

Init$\subseteq S$ is a set of initial states.

AP is a set of atomic propositions.

L: $S \longrightarrow 2^{\text{AP}}$ is a labeling function.

LTS is often drawn as a directed graph with vertices representing the states and edges representing the transitions. Now, we give an example combining CADS in Figure 8.

4.1.5. LTS Example. Consider a part of the CADS model; at the beginning of overtaking, we care about the transition between $s1$: TurnLeft and $s2$: OvertakeReq.

$\text{LTS}_C = (S_C, \text{Act}_C, \longrightarrow_C, \text{Init}_C, \text{AP}_C, L_C)$, where

$$\begin{aligned} &S_C = \{s_1, s_2\}; \text{Act}_C = \{\text{SendReq, waiting}\}; \\ &\longrightarrow_C = \{(s_1, \text{SendReq}, s_2)t)n, q(s_2, \text{waiting}, s_1)\}; \\ &\text{AP}_C = \{\text{Creq} \le 4s\}; L_C = \{s_1 \longrightarrow \varnothing, s_2 \longrightarrow \{\text{Creq} \le 4s\}\}. \end{aligned} \tag{5}$$

According to Definition 7, we give the formal and precise SOS rules for SMD.

Definition 8. Operation semantics of sequence rule

Given source and target states, trigger event, Boolean guard condition, and sequence effect action satisfy that $\{s_s, s_t\} \subseteq S$, $e \in E$, $g \in G$, and $a \in$ Act, if the trigger event occurs and Boolean condition is true, then $s_s \xrightarrow{e(g)/a} s_t$.

4.1.6. Sequence Example. At this point, we give the sequence operation semantics shown in Figure 8. From source state $s7$: OverTaking to $s8$: TurnRight, Boolean guard conditions should be satisfied, and action Acceleration should be performed.

$$\begin{aligned} &\text{PositionOvertaker} - \text{PositionOvertakee} \\ &\ge \text{SafeDistance and } C_{\text{ot}} > 10 \text{ s}. \end{aligned} \tag{6}$$

Definition 9. Operation semantics of choice rule

Choice operation semantics is defined as a function choice $\longrightarrow s_s \times e \times g_i \times a \times s_t, s_s, \quad s_t \in S \wedge e_i \in E \wedge g_i \in G \wedge a_i \in \text{Act} \wedge i \in \mathbb{N}$, where guards G and actions Act are environment signatures for choice rules, guards G are conditions of the transition to happen and can be represented as formulas in some logical language, and actions Act are operations of the block diagram or assignments of attributes.

The choice SOS is defined as follows:

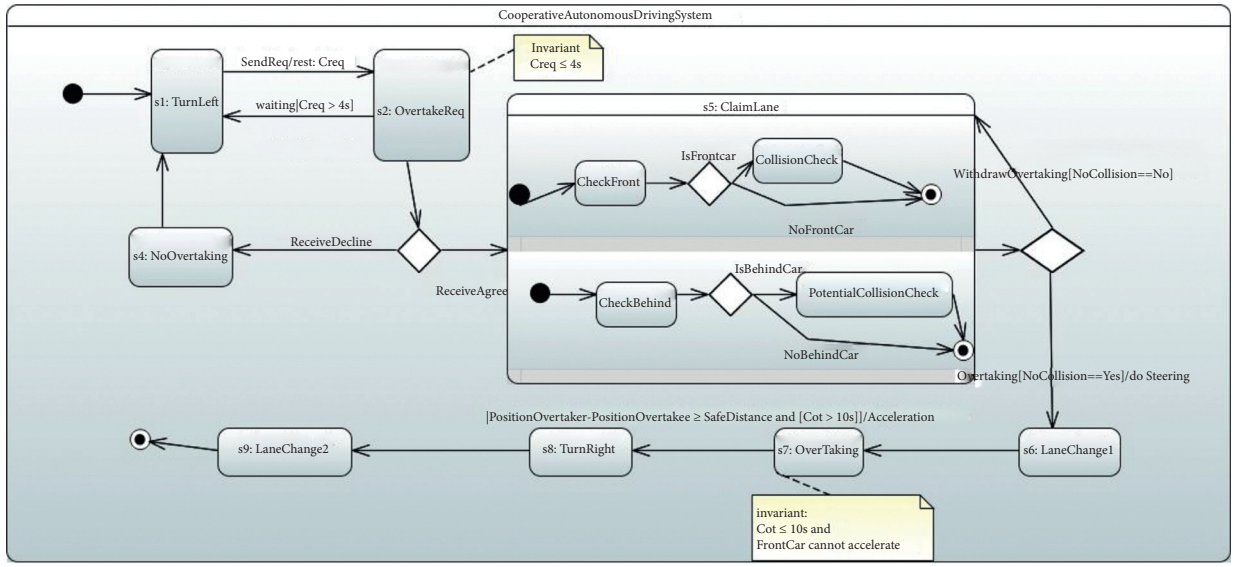

Figure 8: SMD for the autonomous overtaking system.

$$\frac{s_{s1} \xrightarrow{e_1(g_1)/a_1} s_{t1}, s_{s1} \xrightarrow{e_2(g_2)/a_2} s_{t2}, \ldots, s_{s1} \xrightarrow{e_i(g_i)/a_i} s_{ti}, \ldots}{s_{s1} \xrightarrow{\sum_{i\in\mathbb{N}} e_i(g_i)/\sum_{i\in\mathbb{N}} a_i} s_{ti}}. \quad (7)$$

4.1.7. Choice Example. According to Figure 8, there exist four choice transitions, and we choose $s_s = s5, s_t = s5$ or $s6$ to explain choice operation semantics. When event and condition
e_1 = WithdrawOvertaking$\wedge g_1$ = [NoCollision == No] are satisfied, the transition $s5 \longrightarrow s5$ will occur. When another event and guard condition e_2 = Overtaking $\wedge\, g_2$ = [NoCollision == Yes]$\wedge a_2$ = Steering are satisfied, the transition $s5 \longrightarrow s6$ will occur.

Definition 10. Operation semantics of fork rule

A fork vertex splits the incoming transition into two or more terminating transitions belonging to orthogonal target vertices. We can define the fork operation semantics as follows: fork $\longrightarrow s_s \times e \times g \times a \times (r_1.s_{1\text{int}} | r_2.s_{2\text{int}} | \ldots | r_n.s_{n\text{int}})$; $r_n.s_{n\text{int}}, n \in \mathbb{N}$, indicates that target states in fork behavior belong to the orthogonal region. Every transition behaves independently, currently, and synchronously. Now, the definition of fork operation semantics is as follows:

$$\frac{s_s \xrightarrow{e(g)/a} r_1.s_{1\text{int}}, s_s \xrightarrow{e(g)/a} r_2.s_{2\text{int}}, \ldots, s_s \xrightarrow{e(g)/a} r_n.s_{n\text{int}}}{s_s \xrightarrow{e(g)/a} r_1.s_{1\text{int}} | r_2.s_{2\text{int}} | \ldots | r_n.s_{n\text{int}}}. \quad (8)$$

4.1.8. Fork Example. Referring to Figure 8, after the ego vehicle receives agreement information during the invariant Creq ≤ 4s, the state will translate into two orthogonal fork states $s5.s_{1\text{int}}$ and $s5.s_{2\text{int}}$ belonging to the composite state $s5$: ClaimLane.

Definition 11. Operation semantics of join rule

A join vertex merges more than one transition emanating from source vertices belonging to different orthogonal regions and results in synchronous execution. We can define join operation semantics as follows:

$$\text{join} \longrightarrow \left(r_1.s_{1\text{fin}} | r_2.s_{2\text{fin}} | \ldots | r_n.s_{n\text{fin}}\right) \times e \times g \times a \times s_t, \quad (9)$$

where $r_n.s_{n\text{fin}}, n \in \mathbb{N}$, indicates that source states in join behavior belong to the orthogonal region. Source states are from independent, current, and synchronously occurring regions. Now, the definition of join operation semantics is as follows:

$$\frac{r_1.s_{1\text{fin}} \xrightarrow{e(g)/a} s_t, r_2.s_{2\text{fin}} \xrightarrow{e(g)/a} s_t, \ldots, r_n.s_{n\text{fin}} \xrightarrow{e(g)/a} s_t}{r_1.s_{1\text{fin}} | r_2.s_{2\text{fin}} | \ldots | r_n.s_{n\text{fin}} \xrightarrow{e(g)/a} s_t}. \quad (10)$$

4.1.9. Join Example. Referring to Figure 8, the first region r_1 of $s5$: ClaimLaneis responsible for ego_front_vehicle's collision checking, while the second region r_2 of $s5$: ClaimLaneis responsible for ego_behind_vehicle's collision checking. The join state in Figure 8 is a choice Pseudostate from two orthogonal regions r1 and r2.

According to natural language description for autonomous overtaking requirements, we can derive SMD in a semiformal way to specify multiclock system constraints, as shown in Figure 8.

4.2. Semantics of CCSL and Model Transformation. However, due to the lack of precisely formal semantics, UML SMD suffers from an incessant criticism and should manage the gap between the system specification and the design model validation and eliminate ambiguity. In this section, we transform the design model into the analysis model for further verification. In order to do correct model

transformation, first of all, we give the SOS for the formal analysis model. Referring to Section 3, now we give simple Backus–Naur form for CCSL syntax, which contains CC (clock constraint), CR (clock relation), CE (clock expression), CS (clock specification), and Rop (relation operators).

$$
\begin{aligned}
&< \mathrm{CC} > ::= < \mathrm{CC} > , < \mathrm{CR} > | \langle \mathrm{CR} \rangle | < \mathrm{CR} > \text{if bool} \\
&< \mathrm{CR} > ::= < \mathrm{CS} > < \mathrm{Rop} > < \mathrm{CS} > \\
&< \mathrm{Rop} > ::= < \text{subclock} > | \langle \text{exclusion} \rangle | < \text{coincidence} > | \langle \text{precedence} \rangle \\
&< \mathrm{CS} > ::= < \text{clock} > | < \mathrm{CE} > \\
&< \mathrm{CE} > ::= \text{bool?} < \mathrm{CE} > : < \mathrm{CE} > | < \text{clock} > | !1 | !0 \\
&| < \mathrm{CE} > < \text{await} > \text{natural} | \langle < \mathrm{CE} > < \text{sample} > < \mathrm{CE} \rangle | < \mathrm{CE} > < s\_\text{sample} > < \mathrm{CE} > \\
&| \langle < \mathrm{CE} > < \text{upto} > < \mathrm{CE} \rangle | \langle < \mathrm{CE} > < \text{concat} > < \mathrm{CE} \rangle | \langle < \mathrm{CE} > < \text{union} > < \mathrm{CE} \rangle \\
&| < \mathrm{CE} > < \text{inter} > < \mathrm{CE} > | \langle < \mathrm{CE} > < \text{defer} > < \mathrm{CE} \rangle | \langle < \mathrm{CE} > < \text{sup} > < \mathrm{CE} \rangle | \langle < \mathrm{CE} > < \text{inf} > < \mathrm{CE} \rangle .
\end{aligned}
\tag{11}
$$

According to the simple syntax, we give operation semantics in terms of LTS for the formal modelling language CCSL. Based on Definition 7 LTS, we focus on source and target states of a transition; so, the corresponding transition functions are added: α, β: Tr $\longrightarrow S$; Tr denotes a set of transitions, S denotes a set of states, and α, β denote the source and target state of a transition, respectively. The function λ: Tr $\longrightarrow$ Act is added to denote the action which is responsible for the corresponding transition. When clock constraints are treated as transition systems and put in parallel, their composition can be defined as a synchronized product of labelled transition systems.

Definition 12. (synchronized product). Given n automata Au and are based on LTSs in Definition 7, $\mathrm{Au} = \{\mathrm{Au}_1, \mathrm{Au}_2, \mathrm{Au}_3, \ldots, \mathrm{Au}_n\}$; let a synchronization constraint SC be a subset of the product $\mathrm{Au}_1 \times \mathrm{Au}_2 \times \cdots \times \mathrm{Au}_n$, $\mathrm{SC} \subseteq \mathrm{Au}_1 \times \mathrm{Au}_2 \times \cdots \times \mathrm{Au}_n$, and synchronization constraints are beneficial for capturing semantics of CCSL multiclock operators. For the LTS $\mathrm{Au}_i = (S_i, \mathrm{Act}_i, \mathrm{Tr}_i, \mathrm{Init}_i, \mathrm{AP}_i, L_i)$ and $\alpha_i(\mathrm{Tr}_i) \subseteq S_i, \beta_i(\mathrm{Tr}_i) \subseteq S_i$, synchronized product Γ_i of Au_i over the set SC is defined as follows:

$$
\begin{aligned}
S &= S_1 \times S_2 \times \cdots \times S_n, \\
\mathrm{Init} &= \mathrm{Init}_1 \times \mathrm{Init}_2 \times \cdots \times \mathrm{Init}_n, \\
\mathrm{Tr} &= \{\langle \mathrm{tr}_1, \mathrm{tr}_2, \ldots, \mathrm{tr}_n \rangle \in \mathrm{Tr}_1 \times \mathrm{Tr}_2 \times \cdots \times \mathrm{Tr}_n | \langle \lambda_1(\mathrm{tr}_1), \lambda_2(\mathrm{tr}_2), \ldots, \lambda_n(\mathrm{tr}_n) \rangle \in \mathrm{SC}\}, \\
\lambda(\langle \mathrm{tr}_1, \mathrm{tr}_2, \ldots, \mathrm{tr}_n \rangle) &= \langle \lambda_1(\mathrm{tr}_1), \lambda_2(\mathrm{tr}_2), \ldots, \lambda_n(\mathrm{tr}_n) \rangle .
\end{aligned}
\tag{12}
$$

For the clock $c(i) \in C$ involved in the automaton execution, function $\mathrm{Tr}_i \longrightarrow \{\mathrm{tick}, \mathrm{idle}\}^{|C|}$ defines each transition to associate clock states with the corresponding CCSL operators. Based on the actually ticked clocks, the states update according to the function $S_i \longrightarrow \mathrm{Tr}_i^P$, which will return outgoing transitions for a state.

4.2.1. Example of SOS of Clock Relation. According to the definition of synchronization constraint SC and synchronized product Γ_i, operational semantics of multiclock relations is as follows:

(1) Subclock relation $c_1 \subseteq c_2$ is the synchronized product clock c_1, c_2 and $\mathrm{SC}_S = \{\langle c_1, c_2 \rangle, \langle \phi, c_2 \rangle, \langle \phi, \phi \rangle\}$.

(2) Coincidence relation $c_1 \equiv c_2$ is the synchronized product clock c_1, c_2 and $\mathrm{SC}_{\mathrm{CO}} = \{\langle c_1, c_2 \rangle, \langle \phi, \phi \rangle\}$.

(3) Exclusion relation $c_1 \# c_2$ is the synchronized product clock c_1, c_2 and $\mathrm{SC}_E = \{\langle c_1, \phi \rangle, \langle \phi, c_2 \rangle, \langle \phi, \phi \rangle\}$.

(4) Precedence relation $c_1 \prec c_2$ is the synchronized product clock c_1, c_2 and $\mathrm{SC}_P = \left\{ \begin{array}{c} (\langle c_1, \phi \rangle \wedge \langle c_1, c_2 \rangle \wedge \langle \phi, c_2 \rangle \wedge (|\langle c_1, \phi \rangle| > |\langle c_1, c_2 \rangle|)) \\ \vee (\langle c_1, \phi \rangle \vee \langle c_1, c_2 \rangle \vee \langle \phi, \phi \rangle) \end{array} \right\}$.

(5) Causality relation $c_1 \preccurlyeq c_2$ is the synchronized product clock c_1, c_2 and $\mathrm{SC}_{\mathrm{CA}} = \{\langle c_1, \phi \rangle, \langle c_1, c_2 \rangle, \langle \phi, \phi \rangle\}$.

(6) Union expression $c_0 \triangleq c_1 + c_2$ is represented by the synchronized product clock c_1, c_2 and $\mathrm{SC}_U = \{\langle c_1, c_2, c_0 \rangle, \langle c_1, \phi, c_0 \rangle, \langle \phi, c_2, c_0 \rangle \langle \phi, \phi, \phi \rangle\}$.

(7) Intersection expression $c_0 \triangleq c_1 * c_2$ is represented by the synchronized product clock c_1, c_2 and

$$
\mathrm{SC}_I = \{\langle c_1, c_2, c_0 \rangle, \langle c_1, \phi, \phi \rangle, \langle \phi, c_2, \phi \rangle \langle \phi, \phi, \phi \rangle\}.
\tag{13}
$$

With respect to other clock relations and clock expressions, we can give operation semantics based on state-based LTS. Then, we introduce how to transform the formalized design SMD model into analysis and formal CCSL models and prove that the bisimulation relation is preserved and satisfied in the process of model transformation.

Rule 1: $(\forall s_{\text{smd}} \in S_{\text{smd}}) \Rightarrow ((s_{\text{smd}} \longrightarrow s_{\text{ccsl}}) \wedge t\ (s_{\text{ccsl}} \in S_{\text{ccsl}})$: for every state s_{smd} belonging to the SMD, it will be mapped to a state s_{ccsl} belonging to the state-based CCSL model.

Rule 2: $(\forall e \in E_{\text{smd}}) \Rightarrow ((e \longrightarrow c) \wedge (c \in \lambda(\text{tr}), \text{tr} \in \text{Tr}_{\text{ccsl}}))$: for every trigger event e in the state machine, it will be mapped to a clock in the target state-based CCSL model.

Rule 3: $(\forall g \in G_{\text{smd}}) \Longrightarrow ((g \longrightarrow Tr_{\text{choice.Bool}}) \wedge t\ (\text{Tr}_{\text{choice.BOOL}} \in \text{Tr}_{\text{ccsl}})$: for every guard condition belonging to the state machine, it will be mapped to the Boolean condition in the choice expression in the state-based CCSL transition system.

Rule 4: operation semantics of sequence rule transformation:

$$\begin{aligned}
&\forall s_1, s_2 \in S_{\text{smd}},\\
&e \in E_{\text{smd}},\\
&g \in G_{\text{smd}},\\
&(s_1, e, g, s_2) \in \text{Tr}_{\text{smd}} \Rightarrow\\
&\left(\begin{array}{c} s_1 \longrightarrow s_1', s_2 \longrightarrow s_2', s_1', s_2' \in S_{\text{ccsl}}, e \longrightarrow c, (c \in \lambda(\text{tr}), \text{tr} \in \text{Tr}_{\text{ccsl}}),\\ (g \longrightarrow \text{Tr}_{\text{choice.Bool}}) \wedge (\text{Tr}_{\text{choice.BOOL}} \in \text{Tr}_{\text{ccsl}}) \end{array} \right).
\end{aligned} \tag{14}$$

As for the sequence operation, state, event, and guard of the state machine can be transformed into state, clock, and choice Boolean condition in the state-based CCSL model, respectively.

Rule 5: operation semantics of choice rule transformation:

$$\begin{aligned}
&(\forall S_s, \text{Tr}_i, e_i, g_i, S_1, \text{Tr}_i \in S_{\text{smd}} \wedge i \in \mathbb{N} \wedge e_i \in E \wedge g_i \in G \wedge (S_s, e_i, g_i, \text{Tr}_i) \in Tr_{\text{smd}}) \Rightarrow\\
&\left(\begin{array}{c} S_s \longrightarrow S \wedge \text{Tr}_i \longrightarrow \text{CR.choice} \wedge S, \text{CR.choice} \in S_{\text{ccsl}} \wedge i \in \mathbb{N} \wedge e_i \longrightarrow c_i, (c_i \in \lambda(\text{tr}), \text{tr} \in \text{Tr}_{\text{ccsl}})\\ \wedge (g_i \longrightarrow \text{Tr}_{\text{choice.Bool}}) \wedge (\text{Tr}_{\text{choice.BOOL}} \in \text{Tr}_{\text{ccsl}}) \wedge (S, \text{CR.choice}, c_i, \text{Tr}_{\text{choice.Bool}}) \in \text{Tr}_{\text{ccsl}} \end{array} \right).
\end{aligned} \tag{15}$$

As for one source state in the state machine, there are two or more uncertain target states, which can be transformed into the CCSL model using the clock structure choice operator.

Rule 6: operation semantics of fork rule transformation:

$$\begin{aligned}
&s_s \xrightarrow{e(g)/a} r_1.s_{1\text{int}} | r_2.s_{2\text{int}} | \cdots | r_n.s_{n\text{int}}, \quad n \in \mathbb{N}, s_s, r_{1,2,\ldots,n}, s_{1\text{int},2\text{int},\ldots,n\text{int}} \in S_{\text{smd}},\\
&e \in E, g \in G, a \in \text{Act} \Rightarrow\\
&s_s' \xrightarrow{c} \text{tr}_1' | \text{tr}_2' | \cdots | \text{tr}_n', n \in \mathbb{N}, s_s', \text{tr}_{1,2,\ldots,n}' \in S_{\text{ccsl}}, c \in \lambda(\text{tr}), \text{tr} \in \text{Tr}_{\text{ccsl}}.
\end{aligned} \tag{16}$$

The operation semantics of fork behavior is from one state to more than one state. This is concurrent and independent relation. In the process of fork transformation, concurrent behaviors in state machine orthogonal regions can be mapped to the clock coincidence relation in the CCSL time structure.

Rule 7; operation semantics of join rule transformation:

$$r_1.s_{1\text{fin}}, r_2.s_{2\text{fin}}, \ldots, r_n.s_{n\text{fin}} \xrightarrow{e(g)/a} s_t, \quad n \in \mathbb{N}, r_{1,2,\ldots,n}, s_{1\text{fin},2\text{fin},\ldots,n\text{fin}}, s_t \in S_{\text{smd}}, \\ e \in E, g \in G, a \in \text{Act} \Rightarrow \\ s_1' | s_2' | \ldots | s_n' \xrightarrow{c} s_t, \quad n \in \mathbb{N}, s_{1,2,\ldots,n}', s_t \in S_{\text{ccsl}}, c \in \lambda(\text{tr}), \text{tr} \in \text{Tr}_{\text{ccsl}}. \tag{17}$$

The operation semantics of join behavior is from more than one transition emanating from source vertices belonging to different orthogonal regions to one synchronous target state. In the process of join transformation, the number of more than one concurrent source state can be mapped to the DelayFor $c_0 \triangleq c_1$ clock expression behaviors.

4.3. Proof of Bisimulation Equivalence. After introducing the syntax and semantics of modelling language and performing model transformation, behavioral preservation or trace execution equivalence should be proved. If two transition systems perform the same sequences of actions from initial states, respectively, we treat them as equivalent execution models. There are two necessary and important reasons to compare semantics of two systems through bisimulation: one reason is that if two systems satisfy bisimulation equivalence, we could not distinguish them by behavioral observation. Another reason is that if two systems or processes are bisimulation equivalent, then transitions in the first system can be done and finished in the second system. We note that two systems can simulate each other. We give the definition of bisimulation and proof in Definition 13.

Definition 13. (bisimulation). Let two labelled transition systems $\text{LTS}_1 = (Q_1, \text{Act}, \longrightarrow, \text{Init}_1, \text{AP}_1, L_1)$, $\text{LTS}_2 = (Q_2, \text{Act}, \longrightarrow, \text{Init}_2, \text{AP}_2, L_2)$ and a relation $R \subseteq Q_1 \times Q_2$ between the two labelled transition systems be a bisimulation relation, iff for all $q_1 \in Q_1, q_2 \in Q_2$ such that $q_1 R q_2$ holds; it also holds for all actions $\alpha \in \text{Act}$ that

(1) Whenever $q_1 \xrightarrow{\alpha} q_1'$, then for some $q_2', q_2 \xrightarrow{\alpha} q_2'$ and $q_1' R q_2'$
(2) Whenever $q_2 \xrightarrow{\alpha} q_2'$, then for some $q_1', q_1 \xrightarrow{\alpha} q_1'$ and $q_1' R q_2'$

If two transition systems satisfy the bisimulation definition, we say LTS_1 and $\text{LTS}_2$are bisimular $\text{LTS}_1 \approx R_{b\text{sim}} \text{LTS}_2$, and there exists a bisimulation relation $R_{b\text{sim}}$ (denoted as $\approx$) between them. When the relation $\text{LTS}_1 \approx \text{LTS}_2$ holds, we can draw the following conclusion that all traces of LTS_1 can also be preserved in another transition system LTS_2 and vice versa. Accordingly, the relation $\text{LTS}_1 \approx \text{LTS}_2$ indicates that LTS_1 preserves all possible behaviors of another transition system LTS_2.

Theorem 1. *LTS_1 of SMD (S_{SMD}) and LTS_2 of CCSL S_{CCSL} satisfy bisimulation relation, $S_{SMD} \approx S_{CCSL}$.*

Proof. Let $S_{\text{SMD}} = (Q_1, \text{Act}, \text{Tr}_S, \text{Init}_1, \text{AP}_1, L_1)$ be the LTS of SMD and $S_{\text{CCSL}} = (Q_2, \text{Act}, \text{Tr}_C, \text{Init}_2, \text{AP}_2, L_2)$ be the LTS of CCSL. Let R be a binary relation on $Q_1 \times Q_2, R \subseteq Q_1 \times Q_2$. As for $q_1, q_1' \in Q_1$, $\alpha \in \text{Act}, \text{Tr}_S(q_1, \alpha) = q_1' \quad (q_1 \xrightarrow{\alpha} q_1'); q_2, q_2' \in Q_2, \quad \alpha \in \text{Act}, \text{Tr}_C(q_2, \alpha) = q_2' (q_2 \xrightarrow{\alpha} q_2')$, whenever $q_1 R q_2$ for $(q_1, q_2) \in R$, next we should check the following assertions:

(1) $\forall \alpha \text{Act}, q_1' \in Q_1$. If $q_1 \xrightarrow{\alpha} q_1'$, then there is $q_2' \in Q_2$ such that $q_2 \xrightarrow{\alpha} q_2'$ with $q_1' R q_2'$.
(2) $\forall \alpha \in \text{Act}, q_2' \in Q_2$. If $q_2 \xrightarrow{\alpha} q_2'$, then there is $q_1' \in Q_1$ such that $q_1 \xrightarrow{\alpha} q_1'$ with $q_1' R q_2'$.

Given two behavior preservation transformation transition systems S_{SMD} and S_{CCSL},

(1.1) If $\exists e \in \text{Act}, q_1' \in Q_1, \text{tr}_S = q_1 \times e \times q_1' \in \text{Tr}_S$, according to Definition 12 and transformation rules, $\lambda(q_1') = \{e' | e' \in e^\circ\}$ ($^\circ i$ and i° stand for the predecessor and successor of the i^{th} occurrence in clock C). So, in the model S_{SMD}, events e, e' satisfy $e \prec e'$, and so, in the S_{CCSL} model, there exists a transition $\text{tr}_C = q_2 \times e \times q_2' \in \text{Tr}_C$, and $q_2' = \{e' | e \prec e'\}$; so, we can come to the following conclusion: $\lambda_1(q_1') = \lambda_2(q_2'), (q_1', q_2') \in R$.

(1.2) If $\exists e \in \text{Act}, q_2' \in Q_2, \text{tr}_C = q_2 \times e \times q_2' \in \text{Tr}_C$, according to Definition 12 and transformation rules, $\lambda(q_2') = \{e' | e' \in e^\circ\}$. So, in the model S_{CCSL}, events e, e' satisfy $e \prec e'$, and so, in the S_{SMD} model, there exists a transition $t_S = q_1 \times e \times q_1' \in T_S$, and $q_1' = \{e' | e \prec e'\}$; so, we can come to the following conclusion: $\lambda_2(q_2') = \lambda_1(q_1'), (q_1', q_2') \in R$.

(1.3) If there exists a sequence of events, such as $\langle e, e' \rangle \in E_{\text{seq}}$, $\exists e \in \text{Act}, q_1' \in Q_1, \text{tr}_S = q_1 \times e, e' \times q_1' \in \text{Tr}_S$, according to Definition 12 and transformation rules, $\lambda(q_1') = \{e'' | e'' \in e^\circ \cup e'^\circ\}$. So, in the model S_{SMD}, events satisfy the relation $e'' \prec e$ or $e'' \prec e'$, and so, in the S_{CCSL} model, there exists a transition $\text{tr}_C = q_2 \times e, e' \times q_2' \in \text{Tr}_C$, and $q_2' = \{e'' | e'' \prec e \vee e'' \prec e'\}$; so, we can come to the following conclusion: $\lambda_1(q_1') = \lambda_2(q_2'), (q_1', q_2') \in R$.

(1.4) If there exists a sequence of events, such as $\langle e, e' \rangle \in E_{\text{seq}}$, $\exists e \in \text{Act}, q_2' \in Q_2, \text{tr}_C = q_2 \times e, e' \times q_2' \in \text{Tr}_C$, according to Definition 12 and transformation rules, $\lambda(q_2') = \{e'' | e'' \in e^\circ \cup e'^\circ\}$. So, in the model S_{CCSL}, events satisfy the relation $e'' \prec e$ or $e'' \prec e'$, and so, in the S_{SMD} model, there exists a transition $\text{tr}_S = q_1 \times e, e' \times q_1' \in \text{Tr}_S$, and $q_1' = \{e'' | e'' \prec e \vee e'' \prec e'\}$; so, we can come to the following conclusion: $\lambda_2(q_2') = \lambda_1(q_1'), (q_1', q_2') \in R$.

In conclusion, according to Definition 13 (bisimulation) and the previous proof process, for $\forall q_1 \in Q_1$,

$\alpha \in \text{Act}$, if $q_1 \xrightarrow{\alpha} q_1'$, then $\exists q_2' \in Q_2\, q_2 \xrightarrow{\alpha} q_2'$ and $q_1' R q_2'$ and $\forall q_2 \in Q_2, \alpha \in \text{Act}$, if $q_2 \xrightarrow{\alpha} q_2'$, then $\exists q_1' \in Q_1\, q_1 \xrightarrow{\alpha} q_1'$ and $q_1' R q_2'$. So, the relation R satisfies the relation bisimulation definition, and the two models satisfy bisimulation through defining formal syntax and semantics for source and termination models and model semantic mapping rules. □

5. Case Study

5.1. Autonomous Overtaking Requirement Description for CADS. Although fully autonomous road vehicles (highway, urban traffic, and country roads) without human help/interruption remain futuristic, the automobile industries have been striving to meet these expectations. Academia and industry are committed to improve safety, reliability, and traffic efficiency and ease increasingly congested traffic situations. It is especially imperative for cooperative vehicles' lane-changing and overtaking autonomously in the multilane highway. In this paper, we focus on autonomous driving behavior due to independent and self-sufficient decision-making function. These vehicles can communicate with surrounding relevant vehicles and the infrastructure to perceive traffic information. After understanding these environment perceptions, CADS generate negotiation maneuvers for autonomous lane-changing and overtaking behaviors. Each vehicle involving coordination observes its surrounding traffic and conforms to the generating decision-making to realize overtaking autonomously. In the following, requirements about lane-changing judgement, overtaking procedures, and message exchanging for autonomous overtaking are explained.

We assume that there are three one-way lanes in this paper considering the situation shown in Figure 9; the first lane is reserved_lane (lane 2 in Figure 9) occupied by ego_vehicle, the second lane is applying_lane (lane 3 in Figure 9) claimed to overtake by ego_vehicle, and the third lane is the right side of adjacent_lanes (lane 1 in Figure 9). According to the traffic code, we assume that ego_vehicle must claim the left-side adjacent_lane to complete the overtaking action driving on the right-hand side of the road. In the view of the ego_vehicle overtaking scenario, the following vehicles may be involved, and there may be exist three vehicles (ego_vehicle A, ego_front_vehicle B, and ego_behind_vehicle C) in the currently reserved_lane, two vehicles (applying_lane_behind_vehicle D and applying_lane_front_vehicle E) in the applying_lane, and one vehicle (adjacent_lane_vehicle F) in the adjacent_lane. When ego_vehicle on the currently occupied road has intention to overtake, the first is to turn on the left-turn signal to remind surrounding view vehicles of the lane-change intention. At this moment, there are three safety-critical matters to confirm; firstly, CADS should confirm whether vehicles in the applying_lane are within a safe distance; secondly, after the lane-change signal is observed by them, ego_behind_vehicle within the view cannot accelerate; and the third thing is to send an overtaking message to ego_front_vehicle in the same reserved_lane. After confirmation of overtaking in the first step, overtaking negotiation stage is followed. Ego_front_vehicle needs to reply agree or disagree within a certain time (response time, we assume 4 seconds) after receiving the overtaking message. If ego_vehicle cannot receive the reply message from ego_front_vehicle exceeding response time or receives the disagree message, it should not implement lane-changing overtaking behavior and need to keep the previous driving state. If safety distance constraints are not satisfied, overtaking cannot be implemented too. Only when two constraints (receiving agree message and safety distance) are satisfied at the same time, ego_vehicle can start to perform the first lane-changing action. Overtaking behavior can be decomposed into two-times lane-change actions. After the first lane-change mentioned above, ego_vehicle should speed up, and ego_front_vehicle replying agree message just now cannot accelerate. When ego_vehicle's spatial position is greater than ego_front_vehicle and, meanwhile, they meet safety distance for the second lane-change action, ego_vehicle can implement the second lane-change. Time interval between two-times lane-changing must not exceed stipulated overtaking_time, such as 10 seconds. After ego_vehicle completes lane-change twice and overtakes ego_front_vehicle, CADS will automatically adjust speed and maintain a safety distance between them.

5.2. CCSL for Autonomous Overtaking System (AOS) Specification. In order to present the multiclock autonomous overtaking system specification, we should show involved clocks during the overtaking process. For specifying convenience, each clock's abbreviation is given. Next, we need to list various clocks: ego_turn_signal: ets, applying_lane_safedistance: asd, ego_behind_no_acceleration: ebna, ego_send_lanechange: esl, ego_front_agree: efa, ego_front_disagree: efd, ego_no_overtaking: eno, ego_front_no_acceleration: efna, and ego_send_overtaken: eso. The functional and timing properties of the AOS are specified as follows:

AOS1: ego_turn_signal is always followed by applying_lane_safedistance and ego_behind_no_acceleration, so the corresponding CCSL constraint is that $\text{ets} \sim (\text{asd} * \text{ebna})$.

AOS2: applying_lane_safedistance and ego_behind_no_acceleration trigger (cause) event ego_send_lanechange. In other words, the slowest clock asd∨ebna is faster than both clocks asd and ebna, and the tick of clock asd∨ebna can trigger the occurrence of clock esl. The corresponding CCSL constraint is that $(\text{asd} \vee \text{ebna}) \preccurlyeq \text{esl}$.

AOS3: ego_front_agree or ego_front_disagree will occur after ego_send_lanechange within a response time of 4 s. The corresponding CCSL constraint is that

$$\text{esl} \prec (\text{efa} + \text{efd}) \text{ delayedFor } 1 \text{ on } c_1$$

where clock c_1 = IdealClk discretized By 4 s.

AOS4: when ego_vehicle receives the reply ego_front_disagree, it must trigger the event ego_no_overtaking simultaneously, and the CCSL constraint is that efd ≡ eno.

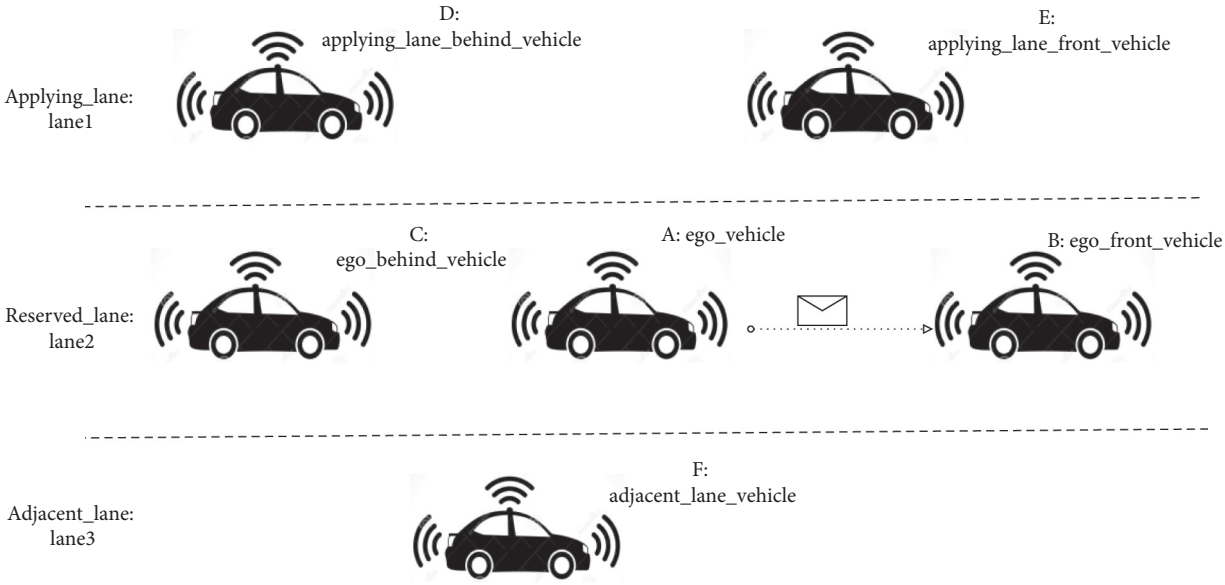

Figure 9: View of the ego-vehicle overtaking.

AOS5: clocks efa, asd, and efna satisfy the constraints (efa∨asd) ≡ efna, which indicates that the slowest event between receiving ego_front_agree message and satisfying applying_lane_safedistance is faster than both of mentioned events, and ego_front_vehicle should trigger the event ego_front_no_acceleration simultaneously.

AOS6: the event ego_send_overtaken will occur with 10 s after the event ego_front_agree. The corresponding CCSL constraint is that

$$\begin{aligned} &\text{efa} \prec \text{eso delayed For 1 on } c_2 \\ &\text{where clock } c_2 = \text{IdealClk discretized By 10 s.} \end{aligned} \tag{18}$$

5.3. Model Validation through Model Simulation and Analysis for the AOS. After the multiclock AOS specification, we can perform model validation through clock constraint simulation intuitively. In the previous part, we present multiclock constraint AOS formal specifications by CCSL, and in this section, clock constraint checking results of model simulation are analyzed using the special tool TimeSquare, which yields a satisfying partial-order execution trace for multiclock constraints. The yielding traces are indicated as VCD format waveforms. When multiclock constraints are correct, TimeSquare will generate a valid simulation trace, but if CCSL specifications do not satisfy all clock constraints or have conflicts, the tool cannot execute and will result in a deadlock shown as the end of VCD waveform. One of the simulation traces is partially shown in Figure 10. The blue and dashed arrows stand for the precedence clock relation, whereas red and vertical solid lines stand for the coincidence relation between two synchronous instants.

Corresponding to safety-critical system properties, clock ego_turn_signal ets must happen alternatively with the intersection clock c30 of two clocks asd and ebna. The clock constraint simulation execution results are shown in Figure 10, and CCSL specification and corresponding expressions in simulation tool TimeSquare are shown in Table 4. Clock constraint ets ∼ (asd ∗ ebna)is satisfied, and this safety-critical property is valid. We define the slowest clock c3 which is faster than clocks asd and ebna, and the tick of supremum clock c3 will cause the tick of clock esl; the result of this clock constraint is satisfied and shown as the clock waveform. Other functional properties are also satisfied and shown in the TimeSquare yielding trace; firstly, the clock efna will only tick if the slowest definition clock csup which is faster than clocks asd and efa is ticking synchronously. Secondly, another strictly synchronous clock constraint efd ≡ eno describes two events occurring in coincidence; event ego_front_disagree corresponding to clock efd means another event ego_vehicle cannot change the lane and implements overtaking behavior corresponding to clock eno; similarly, if ego_front_vehicle replies agreement, then it should not accelerate at the same time. As for timing nonfunctional properties, we combine coincidence and precedence mixed constraints. The clock specification $\text{cupp1} = \begin{pmatrix} \text{efa} \prec \text{eso delayed For 1 on } c_2 \\ \text{where clock } c_2 = \text{IdealClk discretized By } d3 \end{pmatrix}$ constraints cupp1 to tick synchronously with $d3^{\text{th}}$ tick of clock eso, which follows a tick of clock efa, and cupp1 is a mixed constraint since eso and efa are asynchronous precedence clock relation in Table 1. It is similar to another DelayFor constraint $\text{cupp} = \begin{pmatrix} \text{esl} \prec (\text{efa} + \text{efd}) \text{ delayedFor 1 on } c_1 \\ \text{where clock } c_1 = \text{IdealClk discretized By } d2 \end{pmatrix}$, so the first time interval between sending and replying communicating message and the second time interval consumed

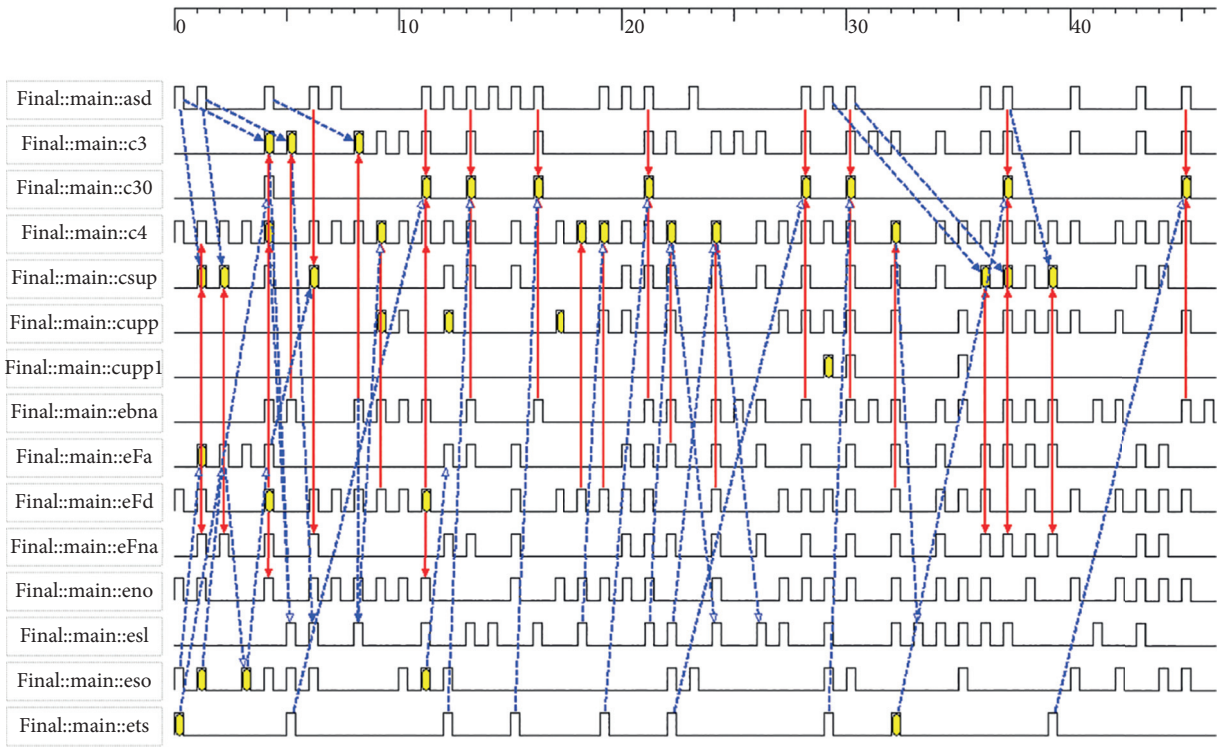

Figure 10: One partial simulation trace for the autonomous overtaking system.

Table 4: CCSL specification and expression in TimeSquare.

CCSL specification	Expression in TimeSquare
ets ~ (asd ∗ ebna)	Expression $c30 = Intersection$(Clock1->*asd*, Clock2->*ebna*) and Relation r1[Alternates](AlternatesLeftClock -> *ets*, AlternatesRightClock -> *c30*)
(asd∨ebna) ≼ esl	Expression $c3 = Sup$(Clock1->*asd*, Clock2->*ebna*) and Relation r2[Causes](LeftClock->c3, RightClock->*esl*)
esl< (efa + efd)delayedFor 1 on c_1 where clock c_1 = IdealClk discretizedBy 4 s	Expression $c4 = Union$(Clock1->*efa*,Clock2->*efd*) and *cupp* = DelayFor(DelayForClockToDelay->*esl*, DelayForClockForCounting->*c4*, DelayForDelay->d2)
efd ≡ eno	Relation r4[Coincides](Clock1->*efd*, Clock2->*eno*)
(efa∨asd) ≡ efna	Expression *csup* = Sup(Clock1->*efa*, Clock2->*asd*) and Relation r5[Coincides](Clock1->*csup*, Clock2->*efna*)
efa<eso delayedFor 1 on c_2 where clock c_2 = IdealClk discretizedBy 10 s	Expression *cupp1*=DelayFor (DelayForClockToDelay->*efa*, DelayForClockForCounting->*eso*, DelayForDelay->d3)

during the process of overtaking are satisfied in the simulation diagram, as shown in Table 4 and in Figure 10.

6. Conclusion and Future Work

In recent years, the emergent multiclock constraint systems, e.g., autonomous driving systems, provide a great promise to our daily life. We are increasingly entrusting our lives to these software-dependent and safety-critical systems. Autonomous systems might seem fraught with danger on account of making individual or collaborative decisions by themselves. Developing multiclock autonomous systems becomes a challenging task because it is urgent and crucial that systems should anticipate the potential collisions, identify aberrant behaviors, and give early warning at the design time before systems' deployment. In this paper, we propose a methodology to bridge the gap between design and analysis for cooperative and highly automated driving systems according to model-driven software development.

Firstly, we have explored the modelling method in multiclock constraint context and presented relevant state-of-the-art specification and verification technologies in the

cooperative autonomous driving domain. We firstly propose autonomous driving domain profile and CADS architecture for a motivation scenario and present the model-based simulation method, domain-specific modelling languages, and dedicated simulator TimeSquare. In order to perform model transformation, we give the syntax and state-based semantics of target multiclock constraint language. Secondly, in the model-driven safety-critical software design, model transformation is a frequently used technique. The operational semantics of both source and terminate languages, i.e., SysML state machine diagram and CCSL, has been formally specified in the form of labelled transition systems. We define strong bisimularity between SMD and CCSL. Behavioral equivalence and semantic preservation have been proven based on semantic mapping rules. Thirdly, through a cooperative autonomous overtaking case study, we use the proposed method to verify safety and reliability at the design phase, and simulation results indicate that the designed system can ensure safe autonomous overtaking driving behavior.

Although this work adopts model-based development and model simulation helping to provide safety evidence, there also exist following limitations. The result of simulation is only one execution trace in terms of presumptions and is strongly dependent on the experience of system safety analysts. Model transformation is executed with a lot of human interactions. Furthermore, we plan to combine formal model checking, which can traverse all software execution traces, or theorem proving techniques to provide rigorous safeguard evidence. At the same time, we aim at developing an automatic transformation tool to realize automatic model transformation under semantic preservation mapping rules. In addition, we are considering to combine artificial intelligence technologies and model-driven software development together to design safe multiclock constraint autonomous systems.

Data Availability

ClockConstraintSystem newfile { imports { //import statements import “platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/kernel.ccslLib” as lib0; import “platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/CCSL.ccslLib” as lib1; } entryBlock main Block main { Relation r1[Alternates](AlternatesLeftClock -~~~~~~~~~~^^^^^^^^^^^^^^^^gt; c1, AlternatesRightClock -~~~~~~~~~~^^^^^^^^^^^^^^^^gt; c2) Clock c1 //-~~~~~~~~~~^^^^^^^^^^^^^^^^gt; evt1(“TheModel:TheClass:p1”): start Clock c2 //-~~~~~~~~~~^^^^^^^^^^^^^^^^gt; evt2(“TheModel:TheClass:p1”): finish } } ClockConstraintSystem newfile { imports { //import statements import “platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/kernel.ccslLib” as lib0; import “platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/CCSL.ccslLib” as lib1; } entryBlock main Block main { Clock c3 Clock c4 Relation r1[Causes](LeftClock-~~~~~~~~~~^^^^^^~~~~~~~~~~gt;c3, RightClock-~~~~~~~~~~^^^^^^~~~~~~gt;c4) } } ClockConstraintSystem newfile { imports { //import statements import “platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/kernel.ccslLib” as lib0; import “platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/CCSL.ccslLib” as lib1; } entryBlock main Block main { Clock c1 Clock c2 Relation r1[Coincides](Clock1-~~~~~~~~~~^^^^^^~~~~~~~~~~gt;c1, Clock2-~~~~~~~~~~^^^^^^~~~~~~~~~~gt;c2) } } ClockConstraintSystem newfile { imports { //import statements import “platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/kernel.ccslLib” as lib0; import “platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/CCSL.ccslLib” as lib1; } entryBlock main Block main { Clock c1 Clock c2 Integer $i = 5$ Integer $j = 6$ Expression c1wait5=Wait(WaitingClock-~~~~~~~~~~^^^^^^~~~~~~~~~~gt;c1, WaitingValue-~~~~~~~~~~^^^^^^~~~~~~~~~~gt; i) Expression c2wait6=Wait(WaitingClock-~~~~~~~~~~^^^^^^~~~~~~~~~~gt;c2, WaitingValue-~~~~~~~~~~^^^^^^~~~~~~~gt;j) Expression myConcat=Concatenation(LeftClock-~~~~~~~~~~^^^^^^~~~~~~~~~~gt;c1wait5, RightClock-~~~~~~~~~~^^^^^^~~~~~~~~~~gt;c2wait6) } } ClockConstraintSystem newfile { imports { //import statements import “platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/kernel.ccslLib” as lib0; import “platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/CCSL.ccslLib” as lib1; import “platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/kernel.ccslLib” as lib0; import “CopyCCSL.ccslLib” as lib1; } entryBlock main Block main { Clock a Clock b Integer p = 5 Relation relation_0 [testConditional](testConditionalLeftClock-~~~~~~~~~~^^^^^^~~~~~~~~~~gt; a, testConditionalRightClock-~~~~~~~~~~^^^^^^~~~~~~~~~~gt; b, testConditionalParam-~~~~~~~~~~^^^^^^~~~~~~~~~~gt; p) } } ClockConstraintSystem newfile { imports { //import statements import “platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/kernel.ccslLib” as lib0; import “platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/CCSL.ccslLib” as lib1; } entryBlock main Block main { //Clock c1 //Clock c2 //Clock c0 //Sequence s1:IntegerSequence= //un=1; //six=6 //(trois=3) //Expression myDefer=Defer(BaseClock-~~~~~~~~~~^^^^^^~~~~~~~~~~gt;c1, DelayClock-~~~~~~~~~~^^^^^^~~~~~~~~~~gt;c2, DelayPatternExpression-~~~~~~~~~~^^^^^^~~~~~~~~~~gt;s1) //Relation r1[Coincides](Clock1-~~~~~~~~~~^^^^^^~~~~~~~~~~gt;c0, Clock2-~~~~~~~~~~^^^^^^~~~~~~~~~~gt;myDefer) Clock a Clock b Clock res23 Sequence s1: IntegerSequence = 1; 3 //(trois=3) Expression myDefer = Defer(BaseClock -~~~~~~~~~~^^^^^^~~~~~~~~~~gt; a, DelayClock -~~~~~~~~~~^^^^^^~~~~~~gt; b, DelayPatternExpression -~~~~~~~~~~^^^^^^~~~~~~~~gt; s1) Relation r1[Coincides](Clock1 -~~~~~~~~~~^^^^^^~~~~~~~~~~gt; res23, Clock2 -~~~~~~~~~~^^^^^^~~~~~~~~~~gt; myDefer) } } ClockConstraintSystem newfile { imports { //import statements import “platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/kernel.ccslLib” as lib0; import “platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/CCSL.ccslLib” as lib1; } entryBlock main Block main { Clock c1 Clock c2 Relation r2[Exclusion](Clock1 -~~~~~~~~~~^^^^^^~~~~~~~~~~gt; c1, Clock2 -~~~~~~~~~~^^^^^^~~~~~~~~~~gt; c2) } } ClockConstraintSystem newfile { imports { //import statements import “platform:/plugin/

fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/kernel.ccslLib" as lib0; import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/CCSL.ccslLib" as lib1; } entryBlock main Block main { Clock c1 Clock c2 //Clock c3 Expression c0=Inf(Clock1-~~~~~~~~~~^^^^^~~~~~~~~~gt;c1, Clock2-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;c2) //Relation r1[Coincides](Clock1-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;c3, Clock2-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;c1Infc2) } } ClockConstraintSystem newfile { imports { //import statements import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/kernel.ccslLib" as lib0; import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/CCSL.ccslLib" as lib1; } entryBlock main Block main { Clock c1 Clock c2 //Clock c3 Expression c0=Intersection(Clock1-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;c1, Clock2-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;c2) //Relation r1[Coincides](Clock1-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;c3, Clock2-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;c1Ic2) } } ClockConstraintSystem periodicitynewfile { imports { //import statements import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/kernel.ccslLib" as lib0; import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/CCSL.ccslLib" as lib1; } entryBlock main Block main { //Clock c1 //Clock res //Integer period=5 //Integer offset=7 //Expression c1Period5offset7=Periodic(PeriodicBaseClock-~~~~~~~~~~^^^^^~~~~~~~~~gt;c1,PeriodicOffset-~~~~~~~~~~^^^^^~~~~~~~~~~gt;offset, PeriodicPeriod -~~~~~~~~~~^^^^^~~~~~~~~~~gt; period) //Relation r1[Coincides](Clock1-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt; res, Clock2 -~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt; c1Period5offset7) Clock c1 Integer period=3 Integer offset=0 Expression c0=Periodic(PeriodicBaseClock-~~~~~~~~~~^^^^^~~~~~~~~~~gt;c1,PeriodicOffset-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt; offset, PeriodicPeriod -~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt; period) } } ClockConstraintSystem newfile { imports { //import statements import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/kernel.ccslLib" as lib0; import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/CCSL.ccslLib" as lib1; } entryBlock main Block main { Clock c1 Clock c2 Relation rp [Precedes](LeftClock-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt; c1, RightClock-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;c2) } } ClockConstraintSystem newfile { imports { //import statements import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/kernel.ccslLib" as lib0; import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/CCSL.ccslLib" as lib1; } entryBlock main Block main { Clock c1 Clock c2 Relation r1[SubClock](LeftClock -~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt; c1, RightClock -~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt; c2) } } ClockConstraint System newfile { imports { //import statements import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/kernel.ccslLib" as lib0; import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/CCSL.ccslLib" as lib1; } entryBlock main Block main { Clock c1 Clock c2 //Clock c3 Expression c0=Sup(Clock1-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;c1, Clock2-~~~~~~~~~~^^^^^~~~~~~~~~gt;c2) //Relation r1[Coincides](Clock1-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;c3, Clock2-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;c1Supc2) } } ClockConstraintSystem newfile { imports { //import statements import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/kernel.ccslLib" as lib0; import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/CCSL.ccslLib" as lib1; } entryBlock main Block main { Clock c1 Clock c2 //Clock c3 Expression c0=Union(Clock1-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;c1, Clock2-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;c2) //Relation r1[Coincides](Clock1-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;c3, Clock2-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;c1Uc2) } } ClockConstraintSystem newfile { imports { //import statements import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/kernel.ccslLib" as lib0; import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/CCSL.ccslLib" as lib1; } entryBlock main Block main { Clock c1 //Clock c2 Integer delay=6 Expression c0=Wait(WaitingClock-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;c1, WaitingValue-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;delay)// Relation r1[Coincides](Clock1-~~~~~~~~~~^^^^^~~~~~~~~~~gt;c2, Clock2-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;c1wait2) } } ClockConstraintSystem 002newfile { imports { //import statements import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/kernel.ccslLib" as lib0; import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/CCSL.ccslLib" as lib1; } entryBlock main Block main { Clock asd Clock ebna Clock ets Clock esl Clock efa Clock efd Clock eno Clock efna Expression c3=Intersection(Clock1-~~~~~~~~~~^^^^^~~~~~~~~~gt;asd, Clock2-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;ebna) Relation r1[Alternates](AlternatesLeftClock -~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt; ets, AlternatesRightClock -~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt; c3) Relation r2[Causes](LeftClock-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;c3, RightClock-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;esl) Expression c4=Union(Clock1-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;efa, Clock2-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;efd) Integer *d*1=1 Integer *d*2=4 Expression clow=DelayFor(DelayForClockToDelay-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;esl, DelayForClockForCounting-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;c4,DelayForDelay-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;d1) Expression cupp=DelayFor(DelayForClockToDelay-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;esl,DelayForClockForCounting-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;c4,DelayForDelay-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;d2) //Relation r3 [Precedes](LeftClock-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt; c4, RightClock-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;c01) Relation r4[Coincides](Clock1-~~~~~~~~~~^^^^^~~~~~~~~~~gt;efd, Clock2-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;eno) Relation r5[Coincides](Clock1-~~~~~~~~~~^^^^^~~~~~~~~~gt;efa, Clock2-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;efna) Clock eso Integer *d*3=10 Expression clow1=DelayFor(DelayForClockToDelay-~~~~~~~~~~^^^^^~~~~~~~~~~gt;eso,DelayForClockForCounting-~~~~~~~~~~^^^^^~~~~~~~~~~~gt;efa,DelayForDelay-~~~~~~~~~~^^^^^~~~~~~~~gt;d1) Expression cupp1=DelayFor(DelayForClockToDelay-~~~~~~~~~~^^^^^~~~~~~~~~~~~~gt;eso, DelayForClockForCounting-~~~~~~~~~~^^^^^~~~~~~~~~~~~~

~~~gt;efa,DelayForDelay-~~~~~~~~~~^^^^^~~~~~~~~~~~~~ ~gt;d3) //Relation r6[Precedes](LeftClock-~~~~~~~~~~^^^^^~~ ~~~~~~~~~gt;eso, RightClock-~~~~~~~~~~^^^^^~~~~~ ~~~~~~gt;c02) } dataTypes{ DenseClockType myPhysicalClock{ baseUnit s physicalMagnitude time }, DenseClockType angle{ baseUnit degree } } }

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was partially supported by the National Natural Science Foundation of China (Grant no. 61772270 and 62077029), by the National Key Research and Development Program (Grant no. 2018YFB1003900), and by the Safety Design and Analysis for Autonomous System Project (Grant no. 9210819807).

References

[1] M. Fisher, L. Dennis, and M. Webster, "Verifying autonomous systems," *Communications of the ACM*, vol. 56, no. 9, pp. 84–93, 2013.

[2] E. Huang, L. F. McGinnis, and S. W. Mitchell, "Verifying SysML activity diagrams using formal transformation to Petri nets," *Systems Engineering*, vol. 23, no. 1, pp. 118–135, 2020.

[3] A. Ferrari, P. Spoletini, and S. Gnesi, "Ambiguity and tacit knowledge in requirements elicitation interviews," *Requirements Engineering*, vol. 21, no. 3, pp. 333–355, 2016.

[4] V. Gervasi, A. Ferrari, D. Zowghi et al., "Ambiguity in requirements engineering: towards a unifying framework," *From Software Engineering to Formal Methods and Tools, and Back*, pp. 191–210, Springer, Cham, Switzerland, 2019.

[5] N. Daclin, S. M. Daclin, V. Chapurlat, and B. Vallespir, "Writing and verifying interoperability requirements: application to collaborative processes," *Computers in Industry*, vol. 82, pp. 1–18, 2016.

[6] M. Ouimet and K. Lundqvist, "Formal software verification: model checking and theorem proving," Technical Report ESL-TIK-00214, Embedded Systems Laboratory, Faridabad, India, 2007.

[7] U. S. Shah and D. C. Jinwala, "Resolving ambiguities in natural language software requirements," *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 5, pp. 1–7, 2015.

[8] D. Harel and B. Rumpe, "Meaningful modeling: what's the semantics of "semantics"?" *Computer*, vol. 37, no. 10, pp. 64–72, 2004.

[9] M. Mori, A. Ceccarelli, P. Lollini et al., "Systems-of-systems modeling using a comprehensive viewpoint-based SysML profile," *Journal of Software: Evolution and Process*, vol. 30, no. 3, p. e1878, 2018.

[10] J. M. Favre, "Towards a basic theory to model model driven engineering," in *Proceedings of the 3rd Workshop in Software Model Engineering, WiSME*, pp. 262–271, Lisbon, Portugal, October 2004.

[11] J. Dyck, H. Giese, and L. Lambers, "Automatic verification of behavior preservation at the transformation level for relational model transformation," *Software & Systems Modeling*, vol. 18, no. 5, pp. 2937–2972, 2019.

[12] N. Gribovskaya and I. Virbitskaite, "Preserving behavior in transition systems from event structure models," in *Proceedings of the CEUR Workshop Proceedings*, p. 2240, Cali, CO, USA, May 2018.

[13] G. J. Uriagereka, E. Amparan, C. M. Martinez et al., "Design-time safety assessment of robotic systems using fault injection simulation in a model-driven approach," in *Proceedings of the 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pp. 577–586, Munich, Germany, September 2019.

[14] S. Kabir, M. Walker, and Y. Papadopoulos, "Dynamic system safety analysis in HiP-HOPS with Petri nets and bayesian networks," *Safety Science*, vol. 105, pp. 55–70, 2018.

[15] M. Luckcuck, M. Farrell, L. A. Dennis, C. Dixon, and M. Fisher, "Formal specification and verification of autonomous robotic systems," *ACM Computing Surveys*, vol. 52, no. 5, pp. 1–41, 2019.

[16] G. V. Bochmann, M. Hilscher, S. Linker, and E.-R. Olderog, "Synthesizing and verifying controllers for multi-lane traffic maneuvers," *Formal Aspects of Computing*, vol. 29, no. 4, pp. 583–600, 2017.

[17] J. Arcile, R. Devillers, and H. Klaudel, "VerifCar: a framework for modeling and model checking communicating autonomous vehicles," *Autonomous Agents and Multi-Agent Systems*, vol. 33, no. 3, pp. 353–381, 2019.

[18] M. Kamali, S. Linker, and M. Fisher, "Modular verification of vehicle platooning with respect to decisions, space and time," in *Proceedings of the International Workshop on Formal Techniques for Safety-Critical Systems*, pp. 18–36, Springer, Gold Coast, Australia, November 2018.

[19] M. Webster, N. Cameron, M. Fisher, and M. Jump, "Generating certification evidence for autonomous unmanned aircraft using model checking and simulation," *Journal of Aerospace Information Systems*, vol. 11, no. 5, pp. 258–279, 2014.

[20] N. Akhtar and M. M. S. Missen, "Contribution to the formal specification and verification of a multi-agent robotic system," *European Journal of Scientific Research*, vol. 117, no. 1, pp. 35–55, 2014.

[21] M. Hilscher and M. Schwammberger, "An abstract model for proving safety of autonomous urban traffic," in *Proceedings of the International Colloquium on Theoretical Aspects of Computing*, pp. 274–292, Springer, Taipei, Taiwan, October 2016.

[22] S. Bernardi, F. Flammini, S. Marrone et al., "Enabling the usage of UML in the verification of railway systems: the DAM-rail approach," *Reliability Engineering & System Safety*, vol. 120, pp. 112–126, 2013.

[23] G.-D. Kapos, A. Tsadimas, C. Kotronis, V. Dalakas, M. Nikolaidou, and D. Anagnostopoulos, "A declarative approach for transforming SysML models to executable simulation models," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, pp. 1–16, 2019.

[24] G. Caltais, S. Leue, and H. Singh, "Correctness of an ATL model transformation from SysML state machine diagrams to promela," in *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development*, pp. 360–372, Valletta, Malta, February 2020.

[25] C. E. Dickerson, R. Roslan, and S. Ji, "A formal transformation method for automated fault tree generation from a UML activity model," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1219–1236, 2018.

[26] B. Alshboul and D. C. Petriu, "Automatic derivation of fault tree models from SysML models for safety analysis," *Journal of*
~~~

Software Engineering and Applications, vol. 11, no. 5, pp. 204–222, 2018.

[27] F. Dias, M. Oliveira, T. Batista et al., "Empowering SysML-based software architecture description with formal verification: from SysADL to CSP," in *Proceedings of the European Conference on Software Architecture*, pp. 101–117, Springer, L'Aquila, Italy, September 2020.

[28] A. Goknil, J. Suryadevara, M. A. Peraldi-Frati et al., "Analysis support for TADL2 timing constraints on EAST-ADL models," in *Proceedings of the European Conference on Software Architecture*, pp. 89–105, Springer, Montpellier, France, July 2013.

[29] B. Chen, X. Li, and X. Zhou, "Model checking of MARTE/CCSL time behaviors using timed I/O automata," *Journal of Systems Architecture*, vol. 88, pp. 120–125, 2018.

[30] J. Suryadevara, C. Seceleanu, F. Mallet et al., "Verifying MARTE/CCSL mode behaviors using UPPAAL," in *Proceedings of the International Conference on Software Engineering and Formal Methods*, pp. 1–15, Springer, Madrid, Spain, September 2013.

[31] C. André and F. Mallet, "Specification and verification of time requirements with CCSL and esterel," in *Proceedings of the ACM SIGPLAN/SIGBED conference on Languages, Compilers, and Tools for Embedded Systems*, pp. 167–176, Dublin, Ireland, June 2009.

[32] L. Huang and E. Y. Kang, "Formal verification of safety & security related timing constraints for a cooperative automotive system," in *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, pp. 210–227, Springer, Prague, Czech Republic, April 2019.

[33] L. Huang, T. Liang, and E. Y. Kang, "Tool-supported analysis of dynamic and stochastic behaviors in cyber-physical systems," in *Proceedings of the IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, pp. 228–239, IEEE, Sofia, Bulgaria, July 2019.

[34] D. Du, P. Huang, K. Jiang, and F. Mallet, "pCSSL: a stochastic extension to MARTE/CCSL for modeling uncertainty in cyber physical systems," *Science of Computer Programming*, vol. 166, pp. 71–88, 2018.

[35] E. Y. Kang and L. Huang, "Probabilistic analysis of timing constraints in autonomous automotive systems using simulink design verifier," in *Proceedings of the International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*, pp. 170–186, Springer, Beijing, China, September 2018.

[36] X. Chen, L. Yin, Y. Yu et al., "Transforming timing requirements into CCSL constraints to verify cyber-physical systems," in *Proceedings of the International Conference on Formal Engineering Methods*, pp. 54–70, Springer, Xi'an, China, November 2017.

[37] V. S. W. Lam and J. Padget, "Analyzing equivalences of UML Statechart diagrams by structural congruence and open bisimulations," in *Proceedings of the IEEE Symposium on Human Centric Computing Languages and Environments*, pp. 137–144, Auckland, New Zealand, October 2003.

[38] B. Tolbi, H. Tebbikh, and H. Alla, "Fault-tolerant continuous flow systems modelling," *International Journal of Systems Science*, vol. 48, no. 1, pp. 107–117, 2017.

[39] J.-P. Bodeveix, M. Filali, M. Garnacho, R. Spadotti, and Z. Yang, "Towards a verified transformation from AADL to the formal component-based language FIACRE," *Science of Computer Programming*, vol. 106, pp. 30–53, 2015.

[40] A. Baouya, D. Bennouar, O. A. Mohamed, and S. Ouchani, "A quantitative verification framework of SysML activity diagrams under time constraints," *Expert Systems with Applications*, vol. 42, no. 21, pp. 7493–7510, 2015.

[41] G. Bacci and M. Miculan, "Structural operational semantics for continuous state stochastic transition systems," *Journal of Computer and System Sciences*, vol. 81, no. 5, pp. 834–858, 2015.

[42] F. Bonchi, T. van Bussel, M. D. Lee, and J. Rot, "Bisimilarity of open terms in stream GSOS," *Science of Computer Programming*, vol. 172, pp. 1–26, 2019.

[43] M. Hülsbusch, B. König, A. Rensink et al., "Showing full semantics preservation in model transformation-a comparison of techniques," in *Proceedings of the International Conference on Integrated Formal Methods*, pp. 183–198, Springer, Nancy, France, October 2010.

[44] A. Narayanan and G. Karsai, "Towards verifying model transformations," *Electronic Notes in Theoretical Computer Science*, vol. 211, pp. 191–200, 2008.

[45] R. Cuer, L. Piétrac, E. Niel, S. Diallo, N. Minoiu-Enache, and C. Dang-Van-Nhan, "A formal framework for the safe design of the autonomous driving supervision," *Reliability Engineering & System Safety*, vol. 174, pp. 29–40, 2018.

[46] W. Do, O. M. Rouhani, and L. Miranda-Moreno, "Simulation-based connected and automated vehicle models on highway sections: a literature review," *Journal of Advanced Transportation*, 2019.

[47] J. DeAntoni and F. Mallet, "Timesquare: treat your models with logical time," in *Proceedings of the International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pp. 34–41, Springer, Vienna, Austria, May 2012.

[48] A. Goknil, J. DeAntoni, M. A. Peraldi-Frati et al., "Tool support for the analysis of TADL2 timing constraints using timesquare," in *Proceedings of the 18th International Conference on Engineering of Complex Computer Systems*, pp. 145–154, IEEE, Singapore, July 2013.

[49] F. Mallet and R. De Simone, "Correctness issues on MARTE/CCSL constraints," *Science of Computer Programming*, vol. 106, pp. 78–92, 2015.

[50] M. M. Morando, Q. Tian, L. T. Truong et al., "Studying the safety impact of autonomous vehicles using simulation-based surrogate safety measures," *Journal of Advanced Transportation*, vol. 2018, Article ID 6135183, 11 pages, 2018.

[51] B. Shen, Z. Zhang, H. Liu et al., "Research on a conflict early warning system based on the active safety concept," *Journal of Advanced Transportation*, vol. 2018, Article ID 8372108, 11 pages, 2018.

[52] Q. Luo, X. Zang, J. Yuan et al., "Research of vehicle rear-end collision model considering multiple factors," *Mathematical Problems in Engineering*, vol. 2020, Article ID 6725408, 11 pages, 2020.

[53] J. Ni, J. Han, and F. Dong, "Multivehicle cooperative lane change control strategy for intelligent connected vehicle," *Journal of Advanced Transportation*, vol. 2020, Article ID 8672928, 10 pages, 2020.

[54] F. Santos, I. Nunes, and A. L. C. Bazzan, "Model-driven agent-based simulation development: a modeling language and empirical evaluation in the adaptive traffic signal control domain," *Simulation Modelling Practice and Theory*, vol. 83, pp. 162–187, 2018.

[55] D. Ameller, X. Burgués, D. Costal, C. Farré, and X. Franch, "Non-functional requirements in model-driven development of service-oriented architectures," *Science of Computer Programming*, vol. 168, pp. 18–37, 2018.

[56] A. P. F. Magalhaes, A. M. S. Andrade, and R. S. P. Maciel, "Model driven transformation development (MDTD): an approach for developing model to model transformation," *Information and Software Technology*, vol. 114, pp. 55–76, 2019.

[57] M. Steurer, A. Morozov, K. Janschek, and K.-P. Neitzke, "SysML-based profile for dependable UAV design," *IFAC-PapersOnLine*, vol. 51, no. 24, pp. 1067–1074, 2018.

[58] N. Kahani, M. Bagherzadeh, J. R. Cordy, J. Dingel, and D. Varró, "Survey and classification of model transformation tools," *Software & Systems Modeling*, vol. 18, no. 4, pp. 2361–2397, 2019.

[59] Z. Wang, G. H. Shen, Z. Q. Huang et al., "A simulation approach for signal time model concern on multi-clock system," in *Proceedings of the National Software Application Conference*, pp. 35–51, Springer, Kunming, China, 2016.

[60] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann, "Polychrony for system design," *Journal of Circuits, Systems and Computers*, vol. 12, no. 3, pp. 261–303, 2003.

[61] L. Besnard, A. Bouakaz, T. Gautier et al., "Timed behavioural modelling and affine scheduling of embedded software architectures in the AADL using polychrony," *Science of Computer Programming*, vol. 106, pp. 54–77, 2015.

[62] Y. Romenska and F. Mallet, "Improving the efficiency of synchronized product with infinite transition systems," in *Proceedings of the International Conference on Information and Communication Technologies in Education, Research, and Industrial Applications*, pp. 285–307, Springer, Kherson, Ukraine, June 2013.

[63] R. Hussain and S. Zeadally, "Autonomous cars: research results, issues, and future challenges," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1275–1313, 2018.