2021 IEEE 29th International Requirements Engineering Conference Workshops (REW) | 978-1-6654-1898-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/REW53955.2021.00076

Environment Model based Requirements Consistency Verification: An Example

Qianqian Liu*, Xiaohong Chen*§, Zhi Jin†‡§
*Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, 200062, China
†Key Laboratory of High-Confidence Software Technologies (Peking University), Ministry of Education, China
‡ Department of Computer Science and Technology, Peking University, Beijing, China
§ Corresponding authors: zhijin@pku.edu.cn, xhchen@sei.ecnu.edu.cn

***Abstract*—Nowadays formal methods have shown their ability in the requirements consistency verification, at least for safety-critical systems. But most requirements verification researches only focus on software requirements specification, without considering the software's interactive environment. In this paper, we propose the environmental properties should be included into the specification verification. They should be considered as inherent constraints that must be satisfied. We extract environmental property constraints from interactive scenarios and state transitions of the environment entities, and transform them into formal models for verification. We use a running example to illustrate the role of environment models in requirements verification.**

***Index Terms*—Environment Modeling based Requirements Engineering, Consistency Verification, Environment Model, Clock Constraint Specification Language**

I. Introduction

Requirements consistency verification is an important but still challenging task especially for safety-critical software systems [1]. Formal methods are strongly recommended here and automated verification is required with the growing scale of software systems. Many formal methods have been applied to the requirements consistency verification [2]–[5] and have shown their ability [6].

We noticed that some requirements inconsistencies can only be revealed by taking into account the properties of the interactive environment, especially when the environment is changing and its behaviors are uncertain. For example, a light unit is about to turn on according to a predefined regime, while there will be a conflict at this moment if an operator presses a button to turn off the light unit. Existing efforts on requirements consistency verification fail to explicitly capture the environmental properties when dealing with the requirements specification.

Our previous work on safety requirements consistency verification [1] has indicated that the environment properties provide essential constraints for ensuring the correct temporal and timing sequences of states and events. This paper explores further the role of environment properties played in consistency verification of software requirements specification. Along the line of the environment modeling based requirements engineering [7], [8], the constraints of the environment on the software are manifested by the permitted interactive scenarios as well as the inherent state transitions of the environment entities. Hence, we propose to extend the scope of the specification verification by including them as the inherent constraints that need to be satisfied.

The rest of the paper is organized as follows. Section II briefly introduces the core concepts of environment modeling based requirements engineering. Section III details how to capture and represent the environmental constraints. Section IV presents the environmental constraint based consistency verification. Section V presents the related work. Section VI concludes the paper and puts forward the future work.

II. Preliminaries

Referring the requirements specification verification, the related environment constraints are represented by the state transition models of the environment entities, i.e., the causal entities, and the scenarios that the environment entities will participant, according to the environment modeling based requirements engineering. This section will briefly introduces these concepts and their structured representation.

Throughout the paper, we use the "light controller problem" as a running example. This problem involves an operator, and a light unit which could be turned on and off by receiving *OnPulse* and *OffPulse*. There are two requirements. One is to control the light unit according to the operators' commands, and the other is to control the light unit by a light regime.

A state diagram describe the behaviors of a causal entity. For example, the light controller problem involves a causal entity, *light unit*, whose state diagram is shown in Fig.1(a). That means a light unit has two states, *On* and *Off*. Event *OnPulse* will trigger the transition from *Off* to *On* while *OffPulse* will trigger the transition from *On* to *Off*.

A scenario diagram represents how a requirement is realized though the interactions between the software and the environment. It contains two kinds of nodes, the behavior interactions (solid line nodes, Beh) and the expected interactions (dotted line nodes, Exp). The former describes the interactions happened between software and the environment entities, while the latter describes the interactions required by the requirements. There are 5 kinds of relations between them [9], [10]. This paper only concerns 4 kinds. The solid arrow from a behavior interaction node to another specifies the behavior order ($BehOrd : Beh \rightarrow Beh$). The dotted arrow from an expected interaction node to another represents the expected order ($ExpOrd : Exp \rightarrow Exp$). The solid line connecting a behavior interaction node and an expected interaction node states that the two nodes connected have the synchronous relation ($Synchronicity : Beh \leftrightarrow Exp$). The arrow from a behavior interaction node to an expected interaction node describes that the behavior interaction enables the expected interaction ($BehEna : Beh \rightarrow Exp$).

Fig.1(b) and (c) show the two scenario diagrams of the light controller problem for realizing the two requirements. Take Fig.1(b) as an example. Firstly, an operator presses OnButton (int_1). Then OnPulse (int_5) is sent to turn on the light unit. This triggers the state of the light unit change to on (int_7) according to the state diagram in Fig.1(a). In this way, the expected interactions int_1 and int_7 are realized. Similarly, OffButton (int_2) enforces OffPulse (int_6), which triggers the state of the light unit change to Off (int_8).

In addition, the scenario diagram also contains some composite nodes to express complex situations, such as *Merge* node, *Branch* node, *Decision* node. *Merge* node specifies the structure where different nodes can trigger one node. *Branch* node specifies the structure where a node can trigger different branching nodes. *Decision* node also represents a branch structure that specifically refers to a trigger of two different branching nodes depending on whether the decision condition is satisfied or not.

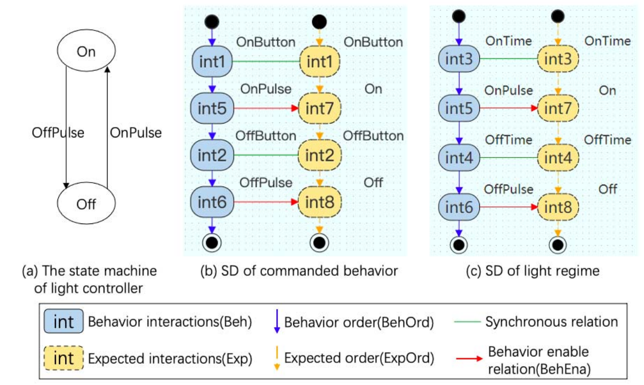


Fig. 1. The environment and requirement models of light controller system

III. Environmental Constraints Capture and Formal Representation

In this section, we present how to capture and formalize the environmental constraints from state diagrams and scenario diagrams. The Clock Constraint Specification Language (CCSL) [11] has been used here with its analysis tools, such as Gra-CCSL [12].

A. CCSL notations

As we know, CCSL is good at describing the causal and temporal relations among events in terms of logic clocks. Each event can be a logical clock which can be defined as a set of ordered instants. Each time the event occurs we say the clock ticks, and the occurrence is an instant of the clock. The i_{th} ($i \in N_{>0}$) instants can be the i_{th} tick of the clock. The event relations can be specified using clock constraints. Here, we only explain intuitive meanings of clock constraints used. Please refer to [11] for more details.

C_1 *StrictPre* ($<$) C_2 means that the i_{th} tick of C_1 occurs strictly before the i_{th} tick of C_2. C_1 *Alternate* ($\sim$) C_2 restricts an alternation of ticks of the left and right clocks. C_1 *Coincidence* ($=$) C_2, restricts that the i_{th} instants of C_1 and C_2 tick together. C_2=C_1 *DelayFor* (\$) d, when C_1 ticks d times, C_2 ticks together with it. C_1=C_2 *Union* (+) C_3 defines a clock C_1 such that C_1 ticks iff C_2 or C_3 ticks. C_1 *Exclude* (#) C_2 restricts that the two clocks never tick together.

These clock constraints can be visualized using a clock graph [13], in which, the nodes are clocks, and edges are clock constraints. Fig.2 gives a clock graph, which contains 6 types of constraints. They are $state_1.s < state_1.f$; $state_1.s \sim state_1.f$; $event_1 = event_2$; $event_2 = state_1.s\$1$; $state_1.f\#state_2.f$; and $s_1.f = state_1.f + state_2.f$.

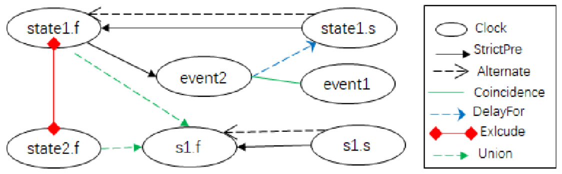


Fig. 2. An example of a clock graph

B. Properties from state diagrams

The state diagram of a causal entity consists of states and state transitions. It is assumed that at any time, the entity situates in only one state. Therefore, the states in a state machine should never be true together. States are mutually exclusive with each other. But the exclusive relation between states can not be directly described in CCSL, because CCSL only deals with events and event relations. In [1], we use two events to represent a state, $state.s$ for the starting of the state, and $state.f$ for the finishing of the state. The exclusive relation of states can be transformed into relations among their start and finish events. Then, we design the state exclusion rule to get the CCSL constraints as shown as R1 in Fig.3.

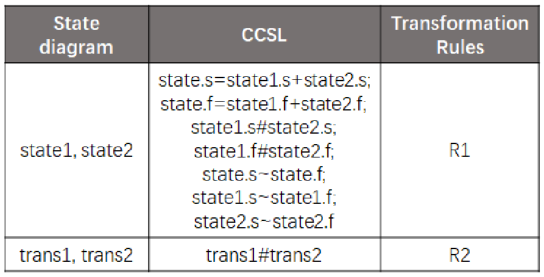
Similarly, multiple transitions in a state diagram cannot occur at the same time. This means they have exclusive relation. As a transition is an event, the exclusive relation between transitions could be directly transformed into the "Exclude" (#) constraints in CCSL as shown as R2 in Fig.3.

C. Environment properties and requirements from scenario diagrams

A scenario diagram specifies the relations among the behavior interactions and expected interactions. According to the relation types, we capture the following relationships.

(1) BehOrd relations

A BehOrd relation in fact specify the execution orders between behavior interactions. According to the structure, we classify them into basic, decision, branch, and loop.



State diagram	CCSL	Transformation Rules
state1, state2	state.s=state1.s+state2.s; state.f=state1.f+state2.f; state1.s#state2.s; state1.f#state2.f; state.s~state.f; state1.s~state1.f; state2.s~state2.f	R1
trans1, trans2	trans1#trans2	R2

Fig. 3. Transformation rules from the state diagram to CCSL constraints

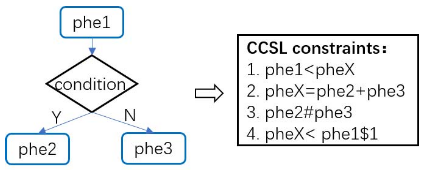
BehOrd_basic: For directly connected interactions (also phenomena as we do not care about the initiator or receiver) in $BehOrd$, i.e., $phe_1 \rightarrow phe_2$, phe_1 happens before phe_2 every time. The relation is in fact "before". It sets the executing trace: phe_1, phe_2, phe_1, phe_2, $\cdots$. From this trace, we can see that phe_1 and phe_2 happens alternatively. As phenomena in the behavior interactions usually are events, so we directly map each phenomenon here to a clock, and this relation could be expressed in CCSL as:

$$phe_1 \sim phe_2$$

This could also be $phe_1 < phe_2$ and $phe_2 < phe_1\$1$.

BehOrd_decision: The *Decision* structure shown in Fig.4 connects a sharing phenomenon and two branching phenomena with a decision condition. If the condition is satisfied after phe_1, then the left branch phe_2 happens. Otherwise, the right branch phe_3 occurs. This means that two branches phe_2 and phe_3 must happen after the share phenomenon phe_1, but they cannot happen together, i.e., they are exclusive.

In CCSL descriptions, we map each phenomenon to a clock, and create a virtual clock phe_X as the union of phe_2 and phe_3, i.e., $phe_X = phe_2 + phe_3$. There should be an Exclude relation between phe_2 and phe_3. Moreover, phe_1 and phe_X also have the same "before" relations with BehOrd_basic. Finally, the CCSL constraints are obtained as shown in Fig.4.

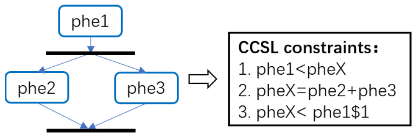


CCSL constraints:
1. phe1<pheX
2. pheX=phe2+phe3
3. phe2#phe3
4. pheX< phe1$1

Fig. 4. BehOrd_decision from scenario diagram to CCSL constraints

BehOrd_branch: The *Branch* structure in behavior order shown in Fig.5 also connects a sharing phenomenon phe_1 and two branching phenomena phe_2 and phe_3. But it has different meaning with decision. After the sharing phenomenon phe_1, phe_2 and phe_3 will happen separately or at the same time. They do not interfere with each other. In other words, phe_2 and phe_3 do not have specific relations.

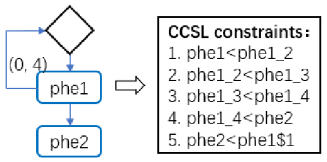
To describe this in CCSL, we also create a virtual clock phe_X as the union of phe_2 and phe_3. We need to designate " phe_1 happens before phe_X". There is no other constraints. So the CCSL constraints are listed in Fig.5.



CCSL constraints:
1. phe1<pheX
2. pheX=phe2+phe3
3. pheX< phe1$1

Fig. 5. BehOrd_branch from scenario diagram to CCSL constraints

BehOrd_loop: There is a phenomenon phe_1 and a phenomenon phe_2 in the $BehOrd$ of scenario diagrams. The control flow starts from the phenomenon phe_1, and after it happens n times (here n is represented by 4), the phenomenon phe_2 occurs, which forms a loop. That is to say, the phenomenon phe_1 triggers the phenomenon phe_2 every 4 times. Moreover, the phenomenon phe_1 can only happen again after the phenomenon phe_2 is triggered. In CCSL descriptions, although phe_1 is one phenomenon, we map it into 4 clocks since it appears 4 times in a loop for simplicity. They all have strictPre relations. At last there should be a strictPre relation for expressing phe_2 is before the next phe_1. The resulting clock constraints are listed in Fig.6.



CCSL constraints:
1. phe1<phe1_2
2. phe1_2<phe1_3
3. phe1_3<phe1_4
4. phe1_4<phe2
5. phe2<phe1$1

Fig. 6. BehOrd_loop from scenario diagram to CCSL constraints

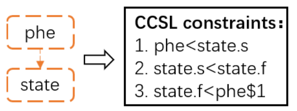
(2) ExpOrd relations

The ExpOrd relations show the sequences of expected interactions. They also have different structures including decision, branch, and loop. Most of the relations are similar with BehOrd, which will be omitted. But we noticed a different kind of nodes in the Exp, i.e., state interactions, which can be identified from the state diagram. For example, in the scenario diagrams of our running example in Fig.1, On and Off are obvious states. States are quite different from events in CCSL constraints. We distinguish two kinds of situations:

ExpOrd_phe2state: As shown in Fig.7(a), phe happens before $state$ and after $state$ happens, the next phe can happen. To express this in CCSL, we map each state into two events, $state.s$ and $state.f$, phe should happen before $state.s$, and $state.f$ must happen before the next phe. The resulting CCSL constraints are listed in Fig. 7(a).

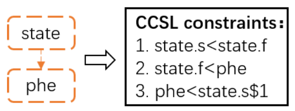
ExpOrd_state2phe: As shown in Fig.7(b), the phenomenon phe happens after $state$. And after phe happens, the next $state$ can happen. Similarly, we map the state into two state events. The finish event $state.f$ must be before the phe, and the start event $state.s$ can only be triggered again after the phenomenon phe happens. Finally the CCSL constraints are shown in Fig. 7(b).

(3) Synchronicity and BehEna relations



CCSL constraints:
1. phe<state.s
2. state.s<state.f
3. state.f<phe$1

(a) $phe \rightarrow state$ structure



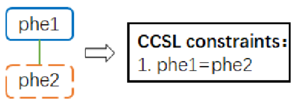
CCSL constraints:
1. state.s<state.f
2. state.f<phe
3. phe<state.s$1

(b) $state \rightarrow phe$ structure

Fig. 7. ExpOrd to CCSL constraints

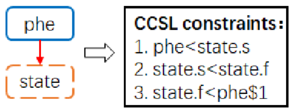
The $Synchronicity$ connecting a behavior interaction to an expected interaction forces the phenomenon phe_1 and phe_2 to happen together. This makes the phenomenon to be a event rather than state. So, in CCSL, we directly use $Coincidence$ relation. The exact description is in Fig.8(a).

The $BehEna$ as shown in Fig.8(b) also connects a behavior interaction to an expected interaction. It means the phe enables or triggers the $state$ to happen. In CCSL descriptions, it is in fact the same with situation $ExpOrd\_phe2state$.



CCSL constraints:
1. phe1=phe2

(a) $synchronicity$ structure



CCSL constraints:
1. phe<state.s
2. state.s<state.f
3. state.f<phe$1

(b) $behEna$ structure

Fig. 8. Synchronicity and BehEna to CCSL constraints

Back to the two scenarios in the running example-light controller problem, we find there are BehOrd_basic, ExpOrd_basic, BehEna, and Synchronicity relations. According to the transforming rules defined above, we get two sets of CCSL constraints as shown in Fig.9.

CCSL constraints:
1. B<OnButton
2. OnButton<OnPulse
3. OnPulse<OffButton
4. OffButton<OffPulse
5. OffPulse<E
6. OnButton<On.s
7. On.s<On.f
8. On.f<OffButton
9. OffButton<Off.s
10. Off.s<Off.f
11. Off.f<E
12. OnPulse<On.s
13. OffPulse<Off.s
14. B~E

CCSL constraints:
1. B<OnTime
2. OnTime<OnPulse
3. OnPulse<OffTime
4. OffTime<OffPulse
5. OffPulse<E
6. OnTime<On.s
7. On.s<On.f
8. On.f<OffTime
9. OffTime<Off.s
10. Off.s<Off.f
11. Off.f<E
12. OnPulse<On.s
13. OffPulse<Off.s
14. B~E

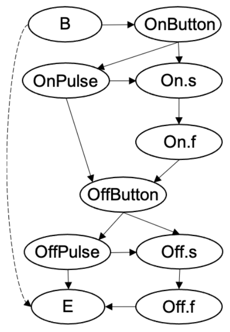
Fig. 9. The CCSL constraints transformed from scenario diagrams in Fig.1

IV. Clock Graph Based Consistency Verification

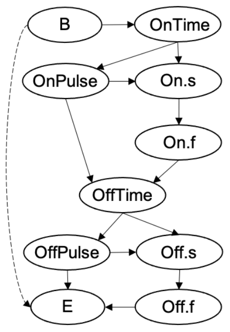
In this section, we firstly compose the clock constraints obtained from scenario diagrams and state diagrams into a clock graph. Then they will be verified using a CCSL analysis tool Gra-CCSL.

A. Clock constraints composition

We design three steps for the composition. Firstly, we visualize the clock constraints obtained from the scenario diagrams into clock graphs. There is direct mapping between CCSL constraints and the clock graph. Fig.10 gives the clock graph of the running example in accordance with CCSL constraints in Fig.9.



(a) Clock graph of Fig.9(a)



(b) Clock graph of Fig.9(b)

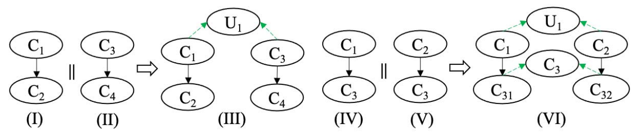
Fig. 10. The clock graphs transformed from CCSL constraints in Fig.9

Secondly, these clock diagrams from the scenarios are composed. Since these scenarios are independent from each, there are parallel relations between them. The parallel relation is in fact "$Union$" constraints in terms of CCSL. By adding it, the sequences between clocks will not change. But we have to consider whether there are shared clocks. Two situations are under consideration:

- There is no shared clock in to-be composed graphs: We only need to create a virtual clock to connect the begin clock of different graphs. For example, in Fig.11(a), clock graph I has two clocks C_1 and C_2 and the constraint $C_1 < C_2$, another clock graph II has two clocks C_3 and C_4 and the constraint $C_3 < C_4$. To ensure the sequences between them, we create a virtual clock U_1 to union C_1 and C_3. Thus the clock graph I and II are composed into clock graph III.
- There are shared clocks in to-be composed graphs: We cannot just simply merge these two clocks into one clock. A shared clock means there is an event happening in each scenario. If we consider the event as a clock C, then the instants happened in scenario A may form a subclock of C, i.e., C_A, and the instants happened in scenario B may form a subclock of C, i.e., C_B, C_A and C_B will union to be C. For example, clock graph IV and V in Fig.11(b), share the same clock C_3, so we create two virtual clocks C_{31} and C_{32} and then union C_{31} and C_{32} with clock C_3 for two different clock graphs to ensure that the orders between the clocks remain unchanged. Thus the clock graph IV and V are composed into clock graph VI.

Finally, we add the clock constraints obtained from the state diagrams to the above clock graph. They are Union, Exclude, and Alternate constraints. One has to cut out the repetition. Then the composition result - a big clock graph is obtained.

We use the light controller example to illustrate the composition. Firstly, we transform the two sets of CCSL constraints transformed from the scenario diagram into a clock graph,



(a) Situation 1 (b) Situation 2

Fig. 11. Clock graph composition situations

then compose all the nodes and edges of the two clock graphs together and remove duplicate edges. After finding the two clock graphs have different begin nodes, we create a virtual clock U to union $OnButton$ and $OnTime$ and add two edges that represent the *Union* relation. Next, we find that the two clock graphs have the same clock $OnPulse$, so we create two virtual clocks $OnPulse_1$ and $OnPulse_2$, and union them with clock $OnPulse$. Similarly, we add the *Union* relations for the clock $OffPulse$, $On.s$, $On.f$, $Off.s$ and $Off.f$. In addition, according to a constraint from the state diagram, e.g., $OnPulse\#OffPulse$, two other constraints will be obtained: $OnPulse_1\#OffPulse_2$ and $OnPulse_2\#OffPulse_1$. Finally, the composed clock graph is shown in Fig.12.

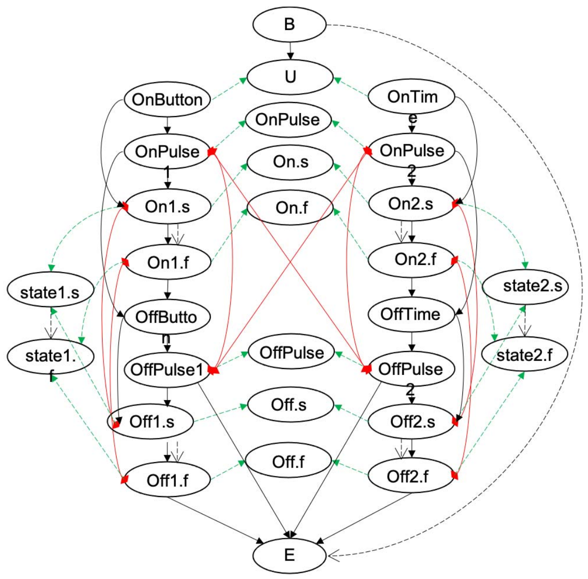


Fig. 12. The composed clock graph of Fig.10

B. Running Gra-CCSL

Gra-CCSL accepts CCSL constraints as input, and use the clock graph to do the verification. Gra-CCSL can find two kinds of inconsistencies, "$hasCyCle$" and "$hasExclude$". "$hasCycle$" inconsistency means there is a cycle of $strictPre$ found in the clock graph, while "$hasExclude$" inconsistency means there are both *Coincidence* and *Exclude* relations between any two nodes in the clock graph.

We input the whole composed clock constraints into Gra-CCSL. If the clock constraints are inconsistent, it will return the least inconsistent clock graph. For example, the result of the running example is as shown in Fig. 13. It is easy to see that the conflict occurs when scenario 1 is sending OnPulse while scenario 2 is sending OffPulse. Otherwise, it will return "pass the check".

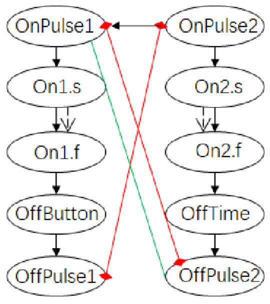


Fig. 13. The clock graph of one inconsistent scenario

V. Related work

There are many researches verifying the consistency of requirements. For example, Langenfeld et al. use the restricted English grammar [14] with the underlying Duration Calculus semantics to specify requirements, and transform it into Phase Event Automata, finally to UPPAAL [15] and Boogie [5]. It uses software model checking and automatic theorem proving to check the consistency of real-time requirements specifications. REQV [4] takes a set of requirements expressed in a structured natural language as input, translates them into LTL and checks their inner consistency with various LTL model checker such as NuSMV. [16] transforms textual requirements specified in Constrained Natural Language(CNL) into the input language of the Z3 SMT solver input. It employs SMT to check the formalized requirements by using the Z3 solver and generates the minimal inconsistent set containing the conflicting requirements. The BTC EMBEDDED PLATFORM tool [17] works on requirements specified on Simplified Universal Pattern(SUP) [18]. The consistency analysis is carried on by requirements coverage and test cases. ReSA [19] is an ontology-based requirements specification language tailored to automotive embedded systems development. It reduces the problem to a boolean satisfiability problem, proposes algorithms for transforming the ReSA specification into boolean expressions, encodes the latter into Z3 assertions, and performs consistency check by using the Z3.

To summarize, the above researches do the consistency verification using formal tools from the early NuSMV to the state-of-the-art Z3. However, environment properties is not utilized. Many researches highlight the role of environment models (domain models) [20]–[22], but the domain models are mainly used for requirements elicitation [7], [23], [24] or formal requirements specification [25], [26], rather than for requirements verification.

Our previous work [1] is also verifying consistency for safety requirements of safety-critical systems. Environment properties are collected manually to provide conflicting events and states, in which the conflicts must be considered to get a correct result. This paper uses state diagrams and scenario diagrams to extract environment properties. Apart from the state and events conflicts, the state diagram can produce a complete conflict set. Moreover, the scenario diagram helps to extract more interactive environmental constraints for the consistency verification.

VI. Conclusion

In this paper, we propose an approach to verify the requirements consistency by considering the constraints from the environment properties. These properties come from the state diagrams of environment entities the describe the internal behavior of the entities, and the scenario diagrams that model the interactions between the software and the environment entities. We provide a method to transform the state diagrams and the scenario diagrams into CCSL models.

As the state diagrams and the scenario diagrams concern about the environment entities and their interactions with the to-be software, our approach is suitable for environment aware systems such as Cyber-Physical systems. However, it is noted that currently, our work only considers a special kind of entity, i.e., the causal entity, the properties of other kind of entities have not been considered yet. We need to further investigate the impacts of the properties of other kinds of environment entities on the requirements inconsistency. We also need study more practical cases.

Acknowledgments

This research was supported by National Key Research and Development Program of China (2018YFB2101300), National Natural Science Foundation of China (61620106007), Shanghai Science and Technology Committee(20ZR1416000).

References

[1] X. Chen, Z. Zhong, Z. Jin, M. Zhang, T. Li, X. Chen, and T. Zhou, "Automating consistency verification of safety requirements for railway interlocking systems," in *27th IEEE International Requirements Engineering Conference, RE2019, Jeju Island, Korea (South), September 23-27, 2019*, 2019, pp. 308–318.

[2] L. Lúcio, S. Rahman, C. Cheng, and A. Mavin, "Just formal enough? automated analysis of EARS requirements," in *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, 2017, pp. 427–434.

[3] A. W. Crapo, A. Moitra, C. McMillan, and D. Russell, "Requirements capture and analysis in ASSERT(TM)," in *25th IEEE International Requirements Engineering Conference, RE, 2017, Lisbon, Portugal, September 4-8, 2017*, 2017, pp. 283–291.

[4] S. Vuotto, M. Narizzano, L. Pulina, and A. Tacchella, "Poster: Automatic consistency checking of requirements with reqv," in *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*, 2019, pp. 363–366.

[5] V. Langenfeld, D. Dietsch, B. Westphal, J. Hoenicke, and A. Post, "Scalable analysis of real-time requirements," in *27th IEEE International Requirements Engineering Conference, RE 2019, Jeju Island, Korea (South), September 23-27, 2019*, 2019, pp. 234–244.

[6] "Formal methods: Foundations and applications - 20th brazilian symposium, sbmf 2017, recife, brazil, november 29 — december 1, 2017, proceedings," ser. Lecture Notes in Computer Science, S. Cavalheiro and J. L. Fiadeiro, Eds., vol. 10623. Springer, 2017.

[7] Z. Jin, *Environment Modeling-Based Requirements Engineering for Software Intensive Systems*. Massachusetts, USA: Morgan Kaufmann, 2018.

[8] Z. Jin, X. Chen, Z. Li, and Y. Yu, "RE4CPS: requirements engineering for cyber-physical systems," in *27th IEEE International Requirements Engineering Conference, RE2019, Jeju Island, Korea (South), September 23-27, 2019*, D. E. Damian, A. Perini, and S. Lee, Eds. IEEE, 2019, pp. 496–497.

[9] Z. Jin, X. Chen, and D. Zowghi, "Performing projection in problem frames using scenarios," in *16th Asia-Pacific Software Engineering Conference, APSEC 2009, 1-3 December 2009, Batu Ferringhi, Penang, Malaysia*. IEEE Computer Society, 2009, pp. 249–256.

[10] X. Chen, B. Yin, and Z. Jin, "DPTool: A Tool for Guiding the Problem Description and the Problem Projection," in *The 18th IEEE International RequirementsEngineering Conference*, 2010, pp. 401–402.

[11] C. André, "Syntax and Semantics of the Clock Constraint Specification Language (CCSL)," INRIA, Research Report RR-6925, 2009.

[12] R. Group, "Gra-CCSL: a clock graph based ccsl constraints analysis tool," 2021. [Online]. Available: https://re4cps/gra-CCSL

[13] F. Mallet, J. Millo, and R. de Simone, "Safe CCSL specifications and marked graphs," in *11th ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMCODE 2013, Portland, OR, USA, October 18-20, 2013*, 2013, pp. 157–166.

[14] A. Post, I. Menzel, and A. Podelski, "Applying restricted english grammar on automotive requirements - does it work? A case study," in *Requirements Engineering: Foundation for Software Quality - Proceedings of 17th International Working Conference (REFSQ 2011)*, 2011, pp. 166–180.

[15] A. Post and J. Hoenicke, "Formalization and analysis of real-time requirements: A feasibility study at BOSCH," in *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings*, 2012, pp. 225–240.

[16] P. Filipovikj, G. Rodríguez-Navas, M. Nyberg, and C. Seceleanu, "Smt-based consistency analysis of industrial systems requirements," in *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*, 2017, pp. 1272–1279.

[17] T. Bienmüller, T. Teige, A. Eggers, and M. Stasch, "Modeling requirements for quantitative consistency analysis and automatic test case generation," in *Workshop on Formal and Model-Driven Techniques for Developing Trustworthy Systems at 18th International Conference on Formal Engineering Methods*, 2016.

[18] J. S. Becker, "Analyzing consistency of formal requirements," *ECEASST*, vol. 76, 2018.

[19] N. Mahmud, C. Seceleanu, and O. Ljungkrantz, "Resa: An ontology-based requirement specification language tailored to automotive systems," in *10th IEEE International Symposium on Industrial Embedded Systems, SIES 2015, Siegen, Germany, June 8-10, 2015*, 2015, pp. 1–10.

[20] M. Broy, "Domain modeling and domain engineering: Key tasks in requirements engineering," in *Perspectives on the Future of Software Engineering, Essays in Honor of Dieter Rombach*, J. Münch and K. Schmid, Eds. Springer, 2013, pp. 15–30.

[21] D. Bjørner, *Software Engineering 3 - Domains, Requirements, and Software Design*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[22] Y. A. Ameur and D. Méry, "Making explicit domain knowledge in formal system development," *Sci. Comput. Program.*, vol. 121, pp. 100–127, 2016.

[23] H. Kaiya and M. Saeki, "Using domain ontology as domain knowledge for requirements elicitation," in *14th IEEE International Conference of Requirements Engineering (RE2006), 11-15 September 2006, Minneapolis/St.Paul, Minnesota, USA*. IEEE Computer Society, 2006, pp. 186–195.

[24] J. Zhou, K. Hänninen, K. Lundqvist, Y. Lu, L. Provenzano, and K. Forsberg, "An environment-driven ontological approach to requirements elicitation for safety-critical systems," in *23rd IEEE International Requirements Engineering Conference, RE2015, Ottawa, ON, Canada, August 24-28, 2015*, D. Zowghi, V. Gervasi, and D. Amyot, Eds. IEEE Computer Society, 2015, pp. 247–251.

[25] S. J. T. Fotso, M. Frappier, R. Laleau, and A. Mammar, "Modeling the hybrid ERTMS/ETCS level 3 standard using a formal requirements engineering approach," *Int. J. Softw. Tools Technol. Transf.*, vol. 22, no. 3, pp. 349–363, 2020.

[26] A. Mammar and R. Laleau, "On the use of domain and system knowledge modeling in goal-based event-b specifications," ser. Lecture Notes in Computer Science, vol. 9952, 2016, pp. 325–339.