Journal of Internet Services and Applications

RESEARCH Open Access

Towards a Verification-Driven Iterative Development of Software for Safety-Critical Cyber-Physical Systems

Marjan Sirjani[1,2*], Luciana Provenzano[1], Sara Abbaspour Asadollah[1], Mahshid Helali Moghadam[1,3] and Mehrdad Saadatmand[3]

*Correspondence:
marjan.sirjani@mdh.se
[1]Mälardalen University, Västerås, Sweden
[2]Reykjavik University, Reykjavik, Iceland
Full list of author information is available at the end of the article

Abstract
Software systems are complicated, and the scientific and engineering methodologies for software development are relatively young. Cyber-physical systems are now in every corner of our lives, and we need robust methods for handling the ever-increasing complexity of their software systems. Model-Driven Development is a promising approach to tackle the complexity of systems through the concept of abstraction, enabling analysis at earlier phases of development. In this paper, we propose a model-driven approach with a focus on guaranteeing safety using formal verification. Cyber-physical systems are distributed, concurrent, asynchronous and event-based reactive systems with timing constraints. The actor-based textual modeling language, Rebeca, with model checking support is used for formal verification. Starting from structured requirements and system architecture design the behavioral models, including Rebeca models, are built. Properties of interest are also derived from the structured requirements, and then model checking is used to formally verify the properties. This process can be performed in iterations until satisfaction of desired properties are ensured, and possible ambiguities and inconsistencies in requirements are resolved. The formally verified models can then be used to develop the executable code. The Rebeca models include the details of the signals and messages that are passed at the network level including the timing, and this facilitates the generation of executable code. The natural mappings among the models for requirements, the formal models, and the executable code improve the effectiveness and efficiency of the approach.

Keywords: Model checking, Verification & validation, Safety-critical systems, Model driven development, Requirements, Cyber-physical systems

1 Introduction

Cyber-physical systems (CPSs) are taking over in our everyday life. In cyber-physical systems, we have embedded computers and networks monitoring and controlling the physical processes. Due to the high interplay between software components and physical processes, in developing software components of such systems we need more robust and rigorous approaches comparing to what is the common practice in software indus-

© The Author(s). 2021 **Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

try today. Moreover, CPSs are commonly safety-critical systems and their failure can have catastrophic consequences on people, environment and facilities.

Verification of safety requirements in cyber-physical systems is a big challenge and of great importance, requiring rigorous solutions. In such systems, due to the existing interactions between the cyber and physical parts over a communication network, concurrency bugs and timing violations may be present. Moreover, in a CPS we deal with asynchrony intrinsic to distributed software systems, and also the alignment of the timeline in the software system and the physical parts. So, we need a modeling framework that supports a proper logical timeline, and is more effective than testing and simulation in discovering timing issues that may impact the safety of the CPS. Indeed, the ability to model *time* is crucial for CPS [1] and there are well-known problems with different concurrency models [2].

Several modeling languages and verification techniques exist and are used for similar purposes and applications. However, actor-based models with timing constraints are suggested as the prime candidates for modeling CPS [3, 4] because the model of computation has less semantic gap with the nature of CPS. Selecting a modeling language that has the least semantic gap with the system being modeled is crucial [5]. Faithfulness (less semantic gap) to the system being modeled and usability for the modeler can be as important as the expressive power of our modeling language.

Formal verification and model checking can be more effective in dealing with concurrency problems comparing to testing and simulation. There is a variety of model checking tools supporting different modeling languages. As an example, the modeling language of Spin [6] is Promela which is a textual process-oriented language. The modeling language supported by NuSMV [7] is a textual form of automata, and UPPAAL [8] is designed to model check timed automata. However, none of these languages are actor-based.

In this paper, we propose an iterative verification-driven development approach for building safety-critical cyber-physical systems using Timed Rebeca. Our approach aims to formally verify time properties for safety requirements in CPS. Timed Rebeca [9–11] is designed for modeling and formal verification of distributed, concurrent and event-driven asynchronous systems with timing constraints. Timed Rebeca is proposed for verification of cyber-physical systems in [12] and is used for modeling and analysing different CPS examples like medical devices [13] and PLCs [14]. Moreover, Rebeca has a textual syntax closer to the target languages for implementation, like C, C++, or Java, which makes Rebeca usable by engineers used to those programming languages without any additional effort[15–17]. So, Rebeca is an optimal choice for our work since it is designed with the purpose of being usable and at the same time analyzable [18].

The novelty of the work is in proposing a light-weight and agile process that covers the life cycle from safety requirements to a formally verifiable abstract code for developing cyber-physical systems. The proposed process helps in identifying ambiguities and inconsistencies in requirements of such systems, which contributes to improve the safety requirements in order to achieve consistency, completeness and correctness.

Our process encompasses different methods, such as the syntax to specify safety requirements and the UML behavioural models, that are common practices in industry. Indeed, our aim is to provide a process that can be usable and used in real industrial settings. From this perspective, this paper describes a *proof of concept* that our approach can work and the transformations among different models can be done smoothly.

The paper is an extension of the conference paper by Sirjani et al [19]. We keep the same prototypical industrial example, i.e., "*Passenger Door Control*", from a train control system as the core of our running example to explain the approach. This example is a simple but prototypical example of a time-critical safety function from a real context. This example represents very well the concurrency and timing issues of a critical safety function that are the main targets of our approach. Here we present the iterative nature of the approach by showing how the requirements, the models and the Rebeca code are improved and extended; and explain the cycles we navigate through the process to debug the code and disambiguate or correct the requirements. We present the mappings in a more structured way, and we go deeper in the model checking exercise.

Structure of the paper. The paper is structured as follows: motivations and background for this work are given in Sections 2 and 3 respectively; in Section 4, the Iterative Verification-Driven Process (VDD-CPS), which is the main contribution of this paper, is introduced; the case study on which our process is applied is presented in Section 5; two additional iterations of the application of our process are provided in Section 6. Related work are discussed in Section 7. Section 8 concludes the paper with some discussions and future work.

2 Motivations

The main motivation behind this work is that timing is an important issue in CPS that may invalidate the safety requirements and, as a result, the safety of the whole system, as explained in Section 1. Moreover, there is a need in industry to be able to verify the safety requirements to guarantee their completeness and correctness before they are implemented, since it is very much costly to fix a safety-critical system built on erroneous safety requirements. This is particularly true for safety-critical CPS where the huge number of components and interactions along with the different disciplines involved make the requirements complex to understand and validate. From this perspective, formal verification represents a valuable candidate to verify safety requirements at an early stage of the development process.

To this aim, the iterative and incremental process for Verification-Driven Development of Cyber-Physical Systems (VDD-CPS), that we discuss in this paper, presents the following characteristics:

- the Rebeca models are built based on the safety requirements specified for the system, and the system architecture. This contributes to build a model that is faithful to the system being modeled. This also implies that the model can be used as the basis for generating the executable code for the system.
- the safety requirements are specified through the GIVEN-WHEN-THEN syntax. This syntax for requirements specification is used in industry to specify requirements at all level of abstraction (i.e. system, sub-system, component requirements), and can be translated into UML models. An advantage of this choice is that this syntax for safety requirements includes some of the core concepts of Rebeca models (refer to Section 5.3), such as actors (i.e. the subject of the condition/s in GIVEN, WHEN, THEN), states (i.e. condition/s in GIVEN), and trigger conditions (i.e. condition/s in WHEN). This makes the mapping of the safety requirements to Rebeca models easy and intuitive, and results in reducing or limiting the errors that may be introduced by

a modeler when creating the Rebeca models, especially when the modeler is not a safety or domain expert. Moreover, this syntax can be easily translated into UML behavioural diagrams, specifically state diagrams and sequence diagrams, which facilitate the transformation of the safety requirements into Rebeca code, as explained in Section 5.3. Another advantage of this choice is that we can build Rebeca models based on real requirements and/or on requirements that are written in the same format as the ones written in real industrial settings. This means providing an approach that can be used in industry to gain formally verified real safety requirements without any additional effort by the engineers.

- the Rebeca models are checked against the safety properties that are also derived from the requirements (refer to Section 5.3), to make sure that the behavioral and implementation details that are added to build the models are not introducing errors.
- the approach is highly iterative to enable not only to build the models but also to refine the requirements on which they are based. Indeed, for building formal models based on requirements, the requirements must be consistent and unambiguous. Otherwise, the models are not correct. The iterative approach goes back and forth to correct and complete the models, and disambiguate, correct and complete the requirements.
- the process is incremental to allow building the Rebeca models by considering only those details of the system that are relevant for the specific iteration. This enables limiting the complexity of the models at each iteration and, as a consequence, avoiding the state space explosion.

It is important to note that the safety requirements in input to our process are elicited through the Safety Requirements Elicitation (SARE) approach [20] (refer to Section 3.1). The main reason why we include this elicitation technique as the starting point of our process, is to ensure that the safety requirements come from the hazards discovered for the system. Often, safety requirements are not correct with respect to the hazards they mitigate due to the fact that the knowledge about the hazards, owned by the safety team, is not always integrated in the requirements. This results in the difficulty to build the right safety architecture [21]. SARE, therefore, gives us the opportunity to obtain the right safety requirements and to iterate also the elicitation, in case errors or omissions in the safety requirements are discovered during the application of our process. However, one can alternatively use the safety requirements from a real system as input to this process. In this case, the SARE approach can be used to complement the safety requirements provided as input or to discover new safety requirements.

3 Background

Here we present an overview of the SARE approach as well as the GIVEN-WHEN-THEN syntax that are used to discover and describe the safety requirements. Also, we introduce Rebeca language that is used to model and verify the code.

3.1 SARE Approach and Structured Requirements

The Safety Requirements Elicitation (SARE) approach proposed in Provenzano et al. [20] is the method we use to elicit the safety requirements that will form the input requirements for the proposed process. The SARE approach exploits the knowledge about hazards acquired during safety analyses as a basis to discover the safety requirements.

This knowledge is stored in the Hazard Ontology, as explained in [22] and [23], and used by the SARE approach to create a list of questions that guide the elicitation of the safety requirements. The resulting requirements are thus "*correct with respect to the hazards they are supposed to mitigate*" [20] since they are elicited based on the knowledge of how hazards occur. Note that the SARE approach can be applied to discover safety requirements at different level of abstractions (e.g. system level, sub-system level, component level), for different types of systems (e.g. individual systems, cyber-physical systems, System of Systems), and for discovering both functional and non-functional requirements (i.e. quality attributes).

To specify the safety requirements elicited by SARE, we use the *GIVEN-WHEN-THEN* syntax in order to obtain well-structured requirements that can be easily used for modeling in Rebeca, The reasons for this choice are explained in Section 2. Specifically, the *GIVEN-WHEN-THEN* is *"a style of specifying a system's behavior using Specification by Example"* [24] developed within the Behavior-Driven Development [25] approach. According to this style, a requirement is decomposed in three parts, the *GIVEN* part states the pre-condition(s) of the action described by the requirement ; the *WHEN* part describes the input event(s) which trigger the action(s); the *THEN* part defines the action(s) the system shall perform as a consequence of the trigger and the expected changes in the system.

Pre-conditions, triggers and actions can be expressed in a language whose vocabulary, syntax and semantics are defined more or less formally. The choice of the language depends on different factors, such as whether the requirements are automatically processed or not, whether the requirements are formally checked or not, whether the requirements are for customers (in this case, a less formal language is more suitable) or technical requirements. This implies that this syntax is suitable to specify requirements at different levels of abstraction (e.g. system level, sub-system level, component level) and at different level of details. Independently of the language chosen, the requirements are structured and all have the three components of pre-conditions, triggers and actions. This makes it easier to write the requirements and facilitates the identification and creation of the appropriate test cases. In this paper, the syntax chosen to express pre-conditions, triggers and action in the GIVEN-WHEN-THEN format is a structured natural language. Specifically, the pre-conditions in *GIVEN* are statements described according to the format "*who is in which state*", where "*who*" can be the system, a sub-system, a component, and so on. For example, in the pre-condition "*the train is ready to run*" of `SafeReq1` in Table 1, "*who*" is "`the train`" and "*in which state*" is "`is ready to run`".

The triggers in *WHEN* are statements described according to the format "*who does what*", where "*who*" can be another system, a component, an external system, and so on, and "*does*" is the verb that describes what occurs. For example, in the trigger "*the driver requests to lock all external doors*" of `SafeReq1` in Table 1, "*who*" is "`the driver`", "*does*" is "`requests to lock`", and "*what*" is "`all external doors`".

Finally, the action in *THEN* is a statement described according to the format "*who shall do~~/be~~ what*", where "*who*" can be the system, a sub-system, a component in charge of doing something or being in a new state, and "*shall do*" describes what shall happen. For example, in the action "*the door controller shall close and lock all the external doors*" of `SafeReq1` in Table 1, "*who*" is "`the door controller`", "*shall do*" is "`shall close and lock`", and "*what*" is "`all external doors`.

Table 1 System safety requirements in GIVEN-WHEN-THEN syntax for the door controller to mitigate the hazard "*Passengers fall out of the train*" connected to the train function "*Open external passenger doors*". These requirements describe the behavior of the external train doors equipped with the lock mechanism that makes the door opening function safer. A slightly revised version of the table in [19]

Name	Safety Requirement
SafeReq1	GIVEN the train is ready to run WHEN the driver requests to lock all external doors THEN the door controller shall close and lock all the external doors
SafeReq2	GIVEN an external door is locked WHEN the passenger requests to open the external door THEN the door controller shall keep the external door closed and locked
SafeReq3	GIVEN an external door is unlocked AND the train is at station WHEN the passenger requests to open an external door THEN the door controller shall open the external door
SafeReq4	GIVEN all external doors on the side of the train close to the platform are unlocked WHEN the driver requests to open all external doors THEN the door controller shall open all external doors on the side of the train close to the platform
SafeReq5	GIVEN the train approaches a station WHEN the driver requests to unlock all external doors that are on the train side close to the platform THEN the door controller shall unlock all external doors on the side of the train close to the platform
SafeReq6	GIVEN the train is running WHEN an external door is open THEN the door controller shall provide an alert

3.2 Timed Rebeca and Verification of Cyber-Physical Systems

The Reactive Object Language, Rebeca [15–17], is an actor-based [26, 27] modeling language supported by theories and tools for formal verification. Rebeca is the first actor-based language with model checking support [18], and is used for modeling and verification of distributed and concurrent systems [5]. The model of computation in Rebeca is event-driven and the communication is asynchronous. The syntax of Rebeca is Java-like. Actors in Rebeca have message queues, each actor takes the message on the top of the queue, execute the method related to that message (called message server) in an atomic and non-preemptive way. While executing a method, messages can be sent to other actors (or itself), and the values of the state variables can change. Sending messages are non-blocking and there is no explicit receive statement.

In Timed Rebeca [10, 11] three keywords are added to model logical time: `delay`, `after` and `deadline`. Time tags are attached to events and states of each actor. Using the keyword `delay`, one can model progress of time while executing a method. If a `send` statement is augmented by `after(t)`, the time tag of the message when it is put in the queue of the receiver is `t` units more than the time tag of the message when it is sent. The time tag of the message when it is sent is the current logical time of the sender. By using `after`, one can model the network delay; periodic events can be modeled using send messages to itself augmented by `after`. The `deadline` keyword models the timeout, if the current time of the receiver actor at the time of triggering the event (taking the message to handle it) is more than the expressed deadline then the model checking tool will complain and raise the deadline-miss warning.

Rebeca is used in different applications, for example in schedulability analysis of wireless sensor network applications [28], protocol verification [29], design exploration and comparing routing algorithms [30].

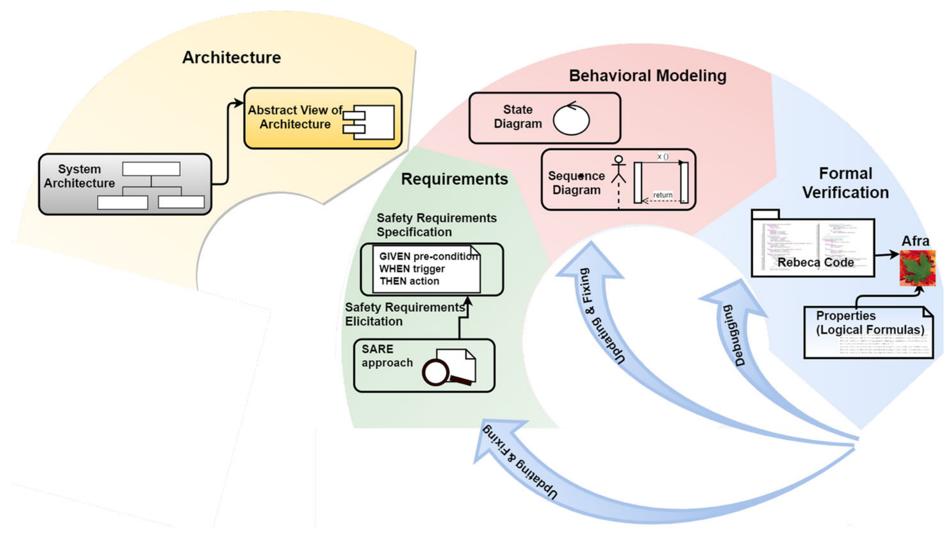


Fig. 1 The iterative and incremental process for Verification-Driven Development of Cyber-Physical Systems (VDD-CPS) from structured requirements to verified Timed Rebeca codes. Formal verification is provided with the model checking tool suite Afra

4 The Iterative Verification-Driven Process: VDD-CPS

The whole process from safety requirements to Rebeca code is depicted in Fig. 1. Specifically, to be able to create the Rebeca code, two inputs are necessary, i.e. the safety requirements and the system architecture. Safety requirements are written in the *GIVEN-THEN-WHEN* syntax [25], as explained in Section 5.1[1]. We distinguish the actors by studying the structured requirements, together with the architecture of the system. Actors are the building blocks of the system and communicate through asynchronous messages or signals. Then we derive the state diagrams from the structured requirements representing the behavior of each actor. We also build a sequence diagram to show the interaction of the actors. Finally, using the state and sequence diagrams we build Rebeca codes. In the process we may discover ambiguities and inconsistencies in any of the artefacts used in the process.

The process of building the Rebeca code from the requirements is an iterative and incremental process, as highlighted by the cycles shown in Fig. 1. The models and the Rebeca code presented in the conference paper by Sirjani et al. [19] can be seen as the first iteration of this process. The current paper addresses the subsequent iterations that aim at improving, by working per increments, both the requirements and the Rebeca code in order to obtain a more complete, unambiguous, and correct set of requirements and a model that best fits them. Nevertheless, the paper is written in a self-contained way and there is no need to first read the conference paper to understand it.

In each iteration, we consider a set of safety requirements and generate the models and the Rebeca code, and then formally verify the safety and progress properties. During each iteration we may find incorrect or ambiguous requirements that show up in the process of building more mathematically-based models. These requirements are updated before a new iteration starts. In each iteration we may consider adding new requirements or pro-

[1] We use this format based on the experience of the second author of the paper who worked for seven years as requirements manager in industry.

perties to check, or changing the Rebeca code to cover more of the existing requirements that are already specified but not yet modelled into Rebeca.

As for now, the Rebeca codes are the final output of our proposed process from safety requirements towards verifiable codes. We can go one step further in the software life cycle and consider producing executable code based on Rebeca. Theatre [31] is an execution platform for Rebeca code. Lingua Franca [32] and its programming model Reactors [33] is another option which targets cyber-physical systems.

In summary in our approach we work on the following artifacts related to the components in Fig. 1:

- System architecture as input to the process (yellow arrow)
- Abstract system architecture built from the system architecture, mapping the architecture components to actors (yellow arrow)
- Safety Requirements (green arrow)
- Structured Requirements (green arrow)
- Behavioral models including UML state diagrams and sequence diagrams (pink arrow)
- Rebeca model (blue arrow)
- Properties of the system based on the requirements represented as logical formula (blue arrow)

The process shown in Fig. 1 includes the transformation of different artifacts and feedbacks in different iterations as follows:

- The mapping from the abstract system architecture and the structured requirements (in Given-When-Then format) as inputs, to behavioral models (UML state diagrams and sequence diagrams) and properties (logical formula) as outputs
- The mapping from the behavioral models to the Rebeca code
- Formal verification of Rebeca code using the model checking tool Afra
- Use the output of the model checking (possible counter examples) to debug the Rebeca model or find further design problems that goes back to the behavioral models or the requirements
- Shorter feedback loops, like finding problems in the requirements while building the behavioral models

In this paper, we focus on the iterative and incremental aspects of our process and present three iterations. In these iterations we incrementally improve and extend both the safety requirements and the Rebeca code. In the first iteration, presented throughout the following sections, the train may be in three different states of *leaving*, *approaching* and *running*. Compared to the version in paper [19], we extended the models to include the *running* state. This way the models are more faithful to the requirements. In the second iteration (Section 6.1) we describe how to manage changes in the Rebeca code to add more details mentioned in the requirements, and in the third iteration (Section 6.2) we show how the process is used to include new safety requirements. One may consider what is presented in paper [19] as iteration zero.

5 The Door Controller Case Study

The case study presented in this paper to exemplify the proposed approach is based on a real industrial case from the railway domain and is chosen based on the experience of the second author in this domain.

We use the function "*Open external passengers doors*" that controls opening of the external doors of a train to let passengers get on and off safely. This function is connected to the hazard '*Passengers fall out of the train*", which is a real hazard for trains and is used to elicit the safety requirements. Specifically, the external doors of a train can be opened by the driver, through a dedicated button installed in the driver's cabin, and by the passenger, through a button placed on each external door. This is done to let passengers get off the train at their destination, and it should be only enabled when the train reaches a station and stops at it. Moreover, the external passenger doors are equipped with a lock mechanism to prevent opening a door when the train leaves the station and is running. This implies that to open a door, the door must be unlocked. This is an interesting function to be modeled and verified for two main reasons:

- The function is safety-related. Indeed, an external door which is accidentally opened when the train is running may cause a passenger to fall out of the train, thus causing an accident.
- The external door can be considered as a shared resource between the driver and the passenger. The door can receive simultaneous commands from the driver (to open, close or lock) and the passenger (to open). This may cause the door to be in an erroneous or unexpected state.

Our aim is therefore to formally check by using the Rebeca modeling language whether there is any possibility that a passenger get off from a running train. In iteration 2 (Section 6.1), we include the information regarding the platforms in the models. In this case, the doors that are on the side of the train opposite to the platform shall be kept locked even when the train is at the station. So, the property to be checked is not only about "getting off from a running train".

It is worth noting that we define "*running*" as the train state which corresponds to the situation where the train is moving between two stations. This means that the train has left the station and is not yet approaching the next one. All external doors are closed and locked. There are multiple properties that can be checked using the Rebeca model checking tool Afra [34], in particular, some of the interesting safety properties that can be checked are the following:

- Is it possible to open a locked door when the train is running?
- Is it possible to open a locked door on the opposite platform when the train is stopping at station?
- Is it possible to open a closed door when the train is ready to leave the station?

Throughout the process we also noticed another interesting scenario that may happen, and the property that has to be checked using model checking:

- Is it possible that a passenger causes a delay in the departure of a train or block it from moving by opening a closed door when the train is ready to leave the station?

Although the safety requirements used in this paper can appear obvious, they are indeed representative of a safety-critical function, i.e. the passenger doors opening while the train is running, whose safety integrity can be violated by timing issues. The safety requirements used in this example is chosen to describe a safety-critical aspect of a train with timing constraints that are difficult to be discovered through testing and simulation and require rigorous verification to guarantee the absence of errors and bugs.

5.1 Safety Requirements of the Example - Initial Input

For the first iteration presented here, we consider the safety requirements elicited by answering the questions in SARE that have been built based on the hazard "*Passengers fall out of the train*". The elicited requirements are then specified in the *GIVEN-WHEN-THEN* syntax, as foreseen by the process (refer to Fig. 1), and according to the syntax described in Section 3.1. The set of safety requirements obtained by performing these two steps is presented in Table 1.

Based on these requirements, we iterate our process in order to remove some ambiguities and remedy the incompleteness. For example, the safety requirement `SafeReq3` in Table 1 is about the passenger being able to open an unlocked door. This requirement is an improved version of an initial version. Model checking reveals that the property of "*a door must not be open when train is running*" fails. A new pre-condition, i.e. "*the train is at station*" is added to prevent the undesirable behaviour. This pre-condition prevents the passenger from opening an unlocked external door when the train is moving. The process of refining the requirements for this specific example is explained in [19].

Another observation made in paper [19] is that most of the concurrency problems in the code are caused because "*close and lock*" (and "*unlock and open*") are not atomic actions. The mechanisms in place to manage the external doors on trains do not guarantee that these actions take place in an atomic way. So, this is a problem that needs to be addressed when writing the software code.

Note that the safety requirements in Table 1 as well as all the safety requirements presented in this work are system safety requirements. We choose to deal with this kind of requirements to start the application of our process from the safety functions that concern the whole system. We think that this helps to grasp a better overview of our approach and of the problem we address in the example. Moreover, being these high-level system requirements less focus on technical details, we think that the example proposed can be easily understood also by readers that are not experienced in the railway domain.

5.2 The System Architecture: Input to the Process

Figure 2 depicts an overview of a typical system architecture realizing the functionalities in our industrial case. The intended system is an example of a cyber-physical system consisting of hardware components like programmable control units, actuators, different communication channels, and different control applications running on the hardware units. The main components in the architecture are Input-Output (I/O) units, central Train Control Unit (TCU), Door Control Unit (DCU). I/O units act as interfaces to the system and are intended to receive/send the input/output signals. The I/O unit on the passenger side are in charge of reading the door push buttons to receive the open request from the passenger. When a passenger pushes the "open" button, the I/O unit receives the open request and sends it to the DCU. The commands for open, close, lock and unlock coming from the driver pass through TCU and go to the DCU. The DCU is responsible for actuating the proper commands for changing the state of the door.

TCU plays the role of the central control management. It might be distributed and run on separate physical devices. For example, one physical control device for running non safety-related functions and one device for the execution of safety-critical functions. DCU may represent a programmable unit which receives the command signal from TCU and applies the signal to the corresponding converters actuating the door. Data communica-

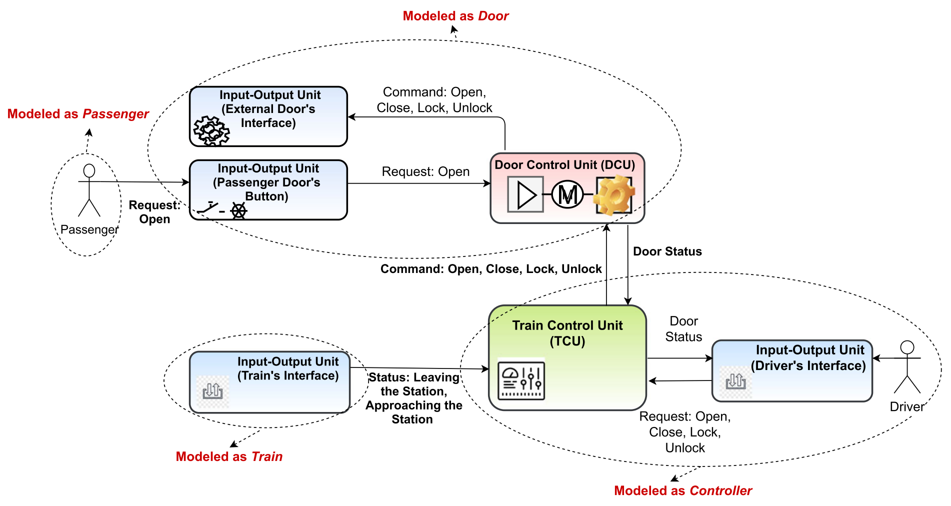

Modeled as *Door*
Modeled as *Passenger*
Input-Output Unit (External Door's Interface)
Command: Open, Close, Lock, Unlock
Input-Output Unit (Passenger Door's Button)
Request: Open
Door Control Unit (DCU)
Passenger
Request: Open
Door Status
Command: Open, Close, Lock, Unlock
Train Control Unit (TCU)
Door Status
Input-Output Unit (Driver's Interface)
Input-Output Unit (Train's Interface)
Status: Leaving the Station, Approaching the Station
Driver
Request: Open, Close, Lock, Unlock
Modeled as *Train*
Modeled as *Controller*

Fig. 2 The system architecture for the door controller system related to the door controller case study from [19]. The dotted circles show the actors in the Rebeca code

tion between the physical devices is usually conducted through a system-wide bus and a safe communication protocol.

Later in our behavioral models, we model both DCU and the associated I/O on the passenger side as `Door` actor and also the combination of TCU and the driver as `Controller` actor. The actor `Train` models a set of I/O units receiving the status from the sensors, and other means, that are used to inform the TCU and the driver that the train is in a state which is significant for our case study, i.e., approached at the station, and ready to leave. These are the states in which the TCU has to change the state of the doors. Figure 2 also shows abstracting the architecture diagram to extract main Rebeca actors.

Generally, in safety critical systems, in order to satisfy the integrity and availability, different types of redundancy structures are applied to different units including I/O units. For example, redundant I/O units are in place and extra supervision mechanisms for the validity check of the resulted values from these redundant I/O units are used. In our example, we abstract these details away. We can create other models focusing on such details and verify the correct functionality of these parts of the system. In general, we need to use compositional and modular approaches to cover large and complicated systems.

5.3 The Transformation Process: Deriving the Behavioral Models and the Rebeca Code

Here we explain how we build the behavioral models based on the requirements. This process is not automated yet and the automation is an ongoing project, as pointed out in the conclusions (refer to Section 8). First we distinguish the actors (or components) in the model that are the building blocks of the system and communicate through asynchronous messages or signals. Then we build the state diagrams for each actor. The state diagram describes the behavior of each actor and how different events change the state of the actors. We also build a sequence diagram to show the interaction of the actors more clearly, and represent the messages and signals passed among the actors. Finally, using the state and sequence diagrams we build Rebeca codes. The final step of this process is mapping Rebeca codes to executable code; in [12] one possible mapping which is building the executable code in Lingua Franca is explained.

Deriving actors. We study the structured requirements, together with the architecture of the system, to distinguish the actors as the building blocks of the model. We build an abstract version of the architecture as a basis for building the behavioral model and subsequently writing the Rebeca code. The abstract architecture includes the actors that will be the reactive classes in the code.

When the system architecture is already in place, our behavioral models, despite of being abstract, are showing the software components that are or will be deployed in the hardware system connected via network. Actors are representing the system components that create events, and react to events. In a pure software system, the architecture can be built based on the requirements and design decisions that may give us more cohesive and decoupled software modules. Here, the components and hence our actors are predetermined based on the system architecture. In an alternative situation, where the system architecture is not already in place, then the approach can be designing the system including the software and hardware from scratch. In that case, we can follow the rules of architecture design in software engineering, or cyber-physical systems engineering, and then we are not restricted to the existing system architecture (hardware and the network). But the outcome should be the same, the actors in the model must represent the components in the system architecture.

Note that only this type of mapping will enable us to check the possible concurrency and timing issues. The model must faithfully capture the components that run concurrently, send signals and messages, and react to events.

Deriving the actors for the train door example. In the context of our door controller example, from the structured requirements (Table 1), we can see that the players are: the *train*, the *driver*, the *passenger*, and the *door*. Note that we do not see the controller in the requirements but it is a central player in the architecture. From the architecture (Fig. 2), we have the I/O units for the passenger door buttons (passing the input to the door to request open) and the door control actuator (passing the output from the door controller to the door, commanding for open, close, lock and unlock (release)). Instead of having an actor representing the passenger button on the door, and another actor representing the door control unit and the actuator, for the sake of simplicity, we model all as one actor *door*.

Another I/O unit is the driver input interface (passing the input to the controller to request open, close, lock and unlock (release)). For simplifying the model, we decided not to model the driver as a standalone actor, the behavior of the driver is merged with the *controller*. We may consider this as an autonomous controller that decides based on the conditions of the doors and the train. We model *train* as an actor to be able to show different states of the train and check the required properties mentioned in the requirements. Passenger is an external entity to the system, but we need to model the inputs from the passenger to check the main safety properties, and hence *passenger* is also an actor. Thus, we need actors to represent the train, the controller, the passenger and the door in the model.

Deriving the states diagrams. We derive the state diagram of each actor based on the explanation in the requirement. From the requirements we see the different states that each actor may be in, and we notice the events that cause the change of states. For the actor that plays the role of a controller the mapping is different. The controller receives the data that indicate changes in the state of other actors, it also receives triggers from environment (sensing). When the controller is notified of certain changes it sends relevant commands to the actors under its control (actuating).

Deriving the state diagrams for the train door example. For the actor train, we consider the states when a train is ready to leave the station, when it is running, and when it approaches the station. When boarding is complete and the *train* is ready to leave, the *driver* sends a request to close and then lock the *doors* and then starts to run. When the *train* approaches the station, the *driver* sends a request to unlock and then open the *doors*. The requests are received by the *controller*, and the controller makes the decision based on the status of the train and the doors. The logic within the code of the *controller* is supposedly written in a way that the safety requirements are guaranteed. There is no exact physical realization as signals or hardware devices for the train in the model, the train is in the model to represent the states where the driver knows she/he has to send the command for closing and locking the doors, or unlocking and opening them.

The *passenger* represents an entity outside the system, and can always request to open the doors. The state diagram of the passenger shows this behavior.

Deriving the sequence diagrams. The process of building the sequence diagrams is similar to building the state diagram, but here the focus is on the messages and signals being passed among the actors. In actors any observable change in the state is caused by an event, so the state diagrams and the sequence diagrams can be checked against each other.

Deriving the states variables. The structured requirements lead to deriving the state variables, and their values, specially the pre- and post-conditions in the *GIVEN* and *THEN* parts. The conditions in the requirements show the states that an actor can be in, we introduce state variables to represent those states. Also, actions explain the changes in the states that need to be captured by state variables. For example, consider the condition "*the train is ready to run*" written in the *GIVEN* part of the requirement `SafeReq1` in Table 1. It shows that we need a variable representing the train status (the variable `trainStatus` of the `Controller` actor in Fig. 5); and one possible value of this variable shows that the train is "*ready to run*". From these requirements, we can also infer that we need two state variables to capture the status of the doors being locked or unlocked, and being opened or closed (the variables `isLocked` and `isClosed` of the `Controller` actor in Fig. 5).

Deriving the events. The events defined in the *WHEN* parts are mapped to the messages that are sent to the actors and upon which the actors react. They can be used to obtain the sequence of messages exchanged among the actors, and to build the sequence diagram.

Deriving the properties. The pre- and post- conditions in the requirements are used to form the assertions that represent the properties to be verified. These conditions show the relation among the derived state variables and we use these specified relations to form the assertions. For instance, consider the requirement `SafeReq2`: "*GIVEN an external door is locked, WHEN the passenger requests to open the external door, THEN the door controller shall keep the external door closed and locked*". This requirement helps us to derive the main safety property of the function "*open external passenger door*". The assertion that shall be checked is: "*It is not possible to open a locked door by passengers*". A stronger assertion that covers this one is discussed in Section 5.5, the assertion is checked by Afra, and we show how the model is modified such that this assertion holds.

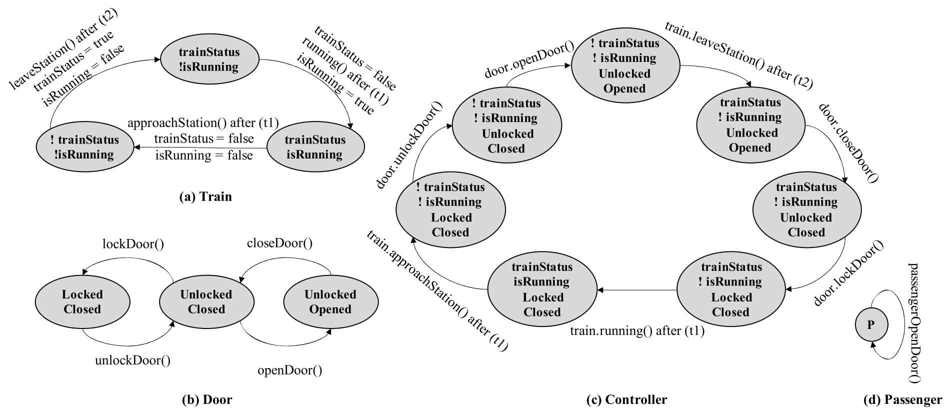


Fig. 3 The state diagrams for the door controller case study. The name of the variables are chosen in a way to make the diagrams self-explanatory as much as possible

There are other interesting requirements, like the requirement `SafeReq4` which is a progress (or liveness) property and shows that progress has to be made. The `SafeReq4` requirement states: "*GIVEN all external doors on the side of the train close to the platform are unlocked, WHEN the driver requests to open all external doors, THEN the door controller shall open all external doors on the side of the train close to the platform*". Safety properties are about showing that nothing bad will happen, while progress properties are about showing that good things will finally happen. The detailed explanation about this requirement and the related property are illustrated in Sections 6.1 and 6.2.

For checking some requirements, we cannot use simple assertions and we need to use the TCTL model checking tool for Timed Rebeca [35][2]. The timing features can be included in TCTL properties, for example for the requirement `SafeReq4`, we can check that "**if** *the doors are unlocked and an open request is sent by the driver* **then** *the doors will be opened within x units of time*". We did not use TCTL model checking in the work presented in this paper.

5.4 The Artifacts: Behavioral Models and the Rebeca Code of the Example

Here we explain the state diagrams, sequence diagrams and the Rebeca code that are derived from the requirements. We also explain the timing properties.

State diagrams. Using the mapping explained in Section 5.3, we can derive the state diagrams for the door controller case study. In Section 5.3, we concluded that we need actors to represent the controller, the door, the driver, the passenger, and the train in the model. Note that we only have one actor that represents all the doors, for the sake of simplicity. The model can be refined, and details can be added in an iterative and incremental way in order to check different properties and different parts of the system.

As shown in the state diagram in Fig. 3a, the train can be in three states: (1) a state when the train has approached the station and stopped (not running), and the passengers leave the train and come on board (!trainStatus & !isRunning); (2) a state when the train is ready to leave, i.e. boarding is completed (trainStatus & !isRunning); (3) a state when the train is running and after some time ready to approach (trainStatus & isRunning). Note that two of the states of the train are important for us in our example because our focus

[2]The TCTL model checking tool for Timed Rebeca is not yet integrated in the Eclipse tool suite of Afra.

is on changing the states of the doors, and we need to change the status of the doors only in these states of the train. For example, when the train is running and door receives an event to open the door the status of the doors should stay unchanged (and that is what the controller in Fig. 3c guarantees by not accepting any *wrong* event in the *wrong* states). The third status is added to show the "*running*" state explicitly to make the behavioral models more faithful to the requirements.

Figure 3b illustrates the states of the doors. A locked and closed door can only be unlocked, and then opened; and an unlocked and open door can only be closed and then locked. The state diagram is consistent with the Rebeca code in Fig. 5. We prevent the door from going to a state where it is locked and open, an unsafe state that should be avoided. The `if-statement` in Line 103 guarantees this.

Figure 3c presents the state diagram for the controller. The controller receives the status of the doors and the train, also the requests for running the train, and opening, closing, locking and unlocking the doors. The controller coordinates the commands that are sent to the doors based on the status of the door itself, and the train.

Figure 3d is the state diagram of the passenger. This actor models the requests coming from the passengers in a non-deterministic way, and the Rebeca code is model checked to make sure this behavior cannot jeopardize the safety.

Sequence diagrams. The sequence diagrams derived from the requirements and the architecture are shown in Figure 4. These diagrams are made in a similar way as described for the state diagram. Indeed, the actors controller, door, passenger and train become the objects in the sequence diagrams among which messages are exchanged in a temporal order to perform the door functions. In the sequence diagrams the flow of messages between actors, and also their order and causality are clearer.

Note that the sequence diagrams are consistent with the Rebeca code. In Fig. 4, it is shown that when the status of the train or the door is changed the controller receives a message to update the status of these two actors in the controller. Any change in the status of the train or the doors triggers the execution of `driveController` message server in which the controller decides which command to send to the train or doors.

The sequence diagram presented in Fig. 4 also shows a Passenger sends the open command directly to the door, and the door sends a message to the controller to update the status in the controller. This is where different errors may occur if the Rebeca code is not written carefully considering the concurrency issues. More explanation is in Section 5.5.

Rebeca code. Based on the state and the sequence diagrams, we wrote a Timed Rebeca code with four reactive classes: `Controller`, `Train`, `Door`, and `Passenger`. The Rebeca code is presented in Fig. 5. The rebecs (i.e. reactive objects, or actors) `controller`, `train`, `door`, and `passenger` are instantiated from these reactive classes.

The main message server of the reactive class `Controller` is `driveController`, where we check the state of the train and the doors, and send proper commands. If the train is in the state that the boarding is completed and the train is ready to run (`trainStatus` is true - lines 35-44), then if the doors are not yet closed, the `Controller` sends a command to close them (by sending the `closeDoor` message to the rebec `door`). If the doors are already closed the controller sends a command to lock them (by sending the `lockDoor` message to the rebec `door`). The message server `DriveController` also checks if the doors are closed and locked then it sends a

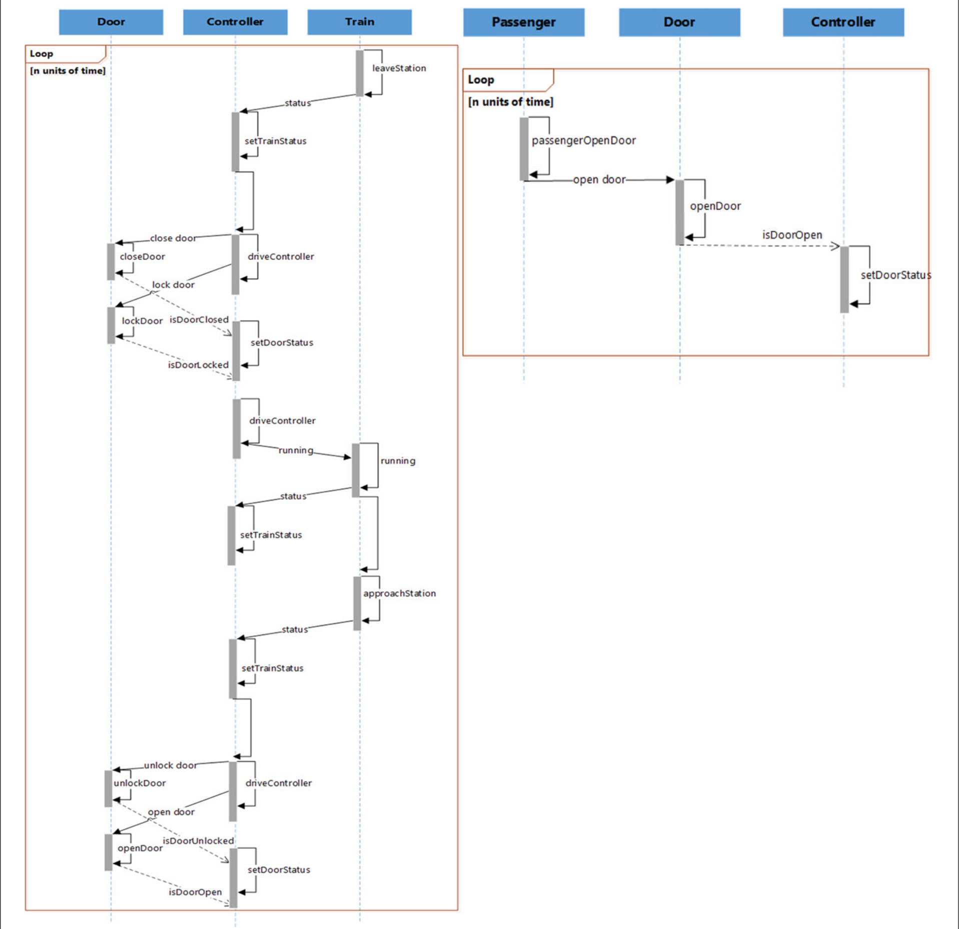


Fig. 4 Sequence diagrams of the door controller case study showing the message passing between the actors `Controller`, `Train`, `Passenger`, and `Door`

command to run the train ((by sending the `running` message to the rebec `train`- lines 42 and 43). If the train is in the approaching state (`trainStatus` is false - lines 45-51), then if the doors are not yet unlocked, the controller sends a command to unlock the doors (by sending the `unlockDoor` message to the rebec `door`). If the doors are already unlocked the controller sends a command to open them (by sending the `openDoor` message to the rebec `door`).

The reactive class `Controller` also has two other message servers: `setDoorStatus` and `setTrainStatus`. The `setDoorStatus` (lines 24-28) is called by the `Door` after updating the status of the doors. The `setTrainStatus` (lines 29-33) is called by the `Train` after updating the status of the train. The reactive class `Train` has three message servers that model the train behavior when the train is ready to leave the station (`leaveStation`), the train is running (`running`) and the train is approaching the station (`approachStation`). The message servers in this actor inform the controller when the train status changes.

The reactive class `Door` models the behavior of the doors and has four message servers: `closeDoor()`, `lockDoor()`, `unlockDoor()` and `openDoor()`. The `closeDoor()` (lines 97-100) is called by `Controller` actor (line 38) to close the door

```
1   env byte networkDelayDoor = 3;
2   env byte networkDelayTrain = 0;
3   env byte reactionDelay = 1;
4   env byte passengerPeriod = 5;
5   env short runningTime = 233;
6   env short atStationTime = 50;
7   reactiveclass Controller(5){
8      knownrebecs{
9        Door door;
10       Train train;
11     }
12     statevars{
13       boolean isClosed;
14       boolean isLocked;
15       boolean trainStatus;
16       boolean commandToMove;
17     }
18     Controller(){
19       trainStatus = true;
20       commandToMove = false;
21       isClosed = false;
22       isLocked = false;
23     }
24      msgsrv setDoorStatus(boolean close, boolean lock) {
25       isClosed = close;
26       isLocked = lock;
27       self.driveController();
28     }
29     msgsrv setTrainStatus(boolean status,
                              Boolean isRunning){
30       trainStatus = status;
31       commandToMove = isRunning;
32       self.driveController();
33     }
34     msgsrv driveController(){
35       if(trainStatus){ // leave the station
36         if(!isClosed || !isLocked) {
37          if(!isClosed)
38            door.closeDoor() after(networkDelayDoor);
39          if(!isLocked)
40            door.lockDoor()
              after(reactionDelay  + networkDelayDoor);
41         }
42         if(isClosed && isLocked && !commandToMove)
43          train.running() after(networkDelayTrain);
44       }// end of if(trainStatus)
45       else if(!trainStatus){ // arrive to the station
46          if(isClosed || isLocked) {
47           if(isLocked)
48            door.unlockDoor() after(networkDelayDoor);
49           if(isClosed)
50             door.openDoor()
               after(reactionDelay  + networkDelayDoor);
51          } } // end of else if(!trainStatus)
52      } // end of driveController()
53     } //end of the Controller class
54  } //end of the Controller class
55  reactiveclass Train(5){
56     knownrebecs{
57       Controller controller;
58     }
59     statevars{
60       boolean status;
61       boolean isRun;
62     }
63     Train(){
64       status = true;
65       isRun = false;
66      self.leaveStation();
67     }
68     msgsrv leaveStation(){
69       status = true;
70       isRun = false;
71       controller.setTrainStatus(status, isRun)
          after(networkDelayTrain);
72     }
73     msgsrv running(){
74       isRun = true;
75       controller.setTrainStatus(status, isRun)
         after(networkDelayTrain);
76       self.approachStation() after(runningTime);
77     }
78     msgsrv approachStation(){
79       status = false;
80       isRun = false;
81       controller.setTrainStatus(status)
         after(networkDelayTrain);
82       self.leaveStation() after(atStationTime);
83     }
84  } //end of the Train class
85  reactiveclass Door(5){
86     knownrebecs{
87       Controller controller;
88     }
89     statevars{
90       boolean isDoorClosed;
91       boolean isDoorLocked;
92     }
93     Door(){
94       isDoorClosed = false;
95       isDoorLocked = false;
96     }
97     msgsrv closeDoor(){
98       isDoorClosed = true;
99       controller.setDoorStatus(isDoorClosed,
         isDoorLocked) after(networkDelayDoor);
100    }
101    msgsrv lockDoor(){
102      if (isDoorClosed){
          // The door is only locked if the door is closed.
103         isDoorLocked = true;
104      }
105      controller.setDoorStatus(isDoorClosed,
         isDoorLocked) after(networkDelayDoor);
106    }
107    msgsrv unlockDoor(){
108     isDoorLocked = false;
109      controller.setDoorStatus(isDoorClosed,
         isDoorLocked) after(networkDelayDoor);
110    }
111    msgsrv openDoor(){
          // The door is only opened if the door is not locked.
112      If (!isDoorLocked){
113         isDoorClosed = false;
114      }
115      controller.setDoorStatus(isDoorClosed,
         isDoorLocked) after(networkDelayDoor);
116    }
117 } //end of the Door class
118 reactiveclass Passenger(5){
119    knownrebecs{
120      Door door;
121    }
122    Passenger(){
123      self.passengerOpenDoor() after(passengerPeriod);
124    }
125    msgsrv passengerOpenDoor(){
126      door.openDoor();
127      self.passengerOpenDoor() after(passengerPeriod);
128    }
129 } //end of the Passenger class
130 main {
131       Controller controller(door, train):();
132       Door door(controller):();
133       Train train(controller):();
134       Passenger passenger(door):();
135       }
```

Fig. 5 The Rebeca code for the door controller case study. This is revised version of the Rebeca code in [19] where the code is adjusted to include the *running* state of the train

by changing the status of the door (line 98). The `lockDoor()` (lines 101-106) is called by the controller (line 40) to lock the door. If the current status of the door is closed, then the status of the door is changed to locked (line 103). The `unlockDoor()` (lines 107-110) is called by the `Controller` actor (line 48) to unlock the door by changing the status of the lock (line 108). The `openDoor()` (lines 111-116) is called by the `Controller` actor (line 50) and the `Passenger` actor (line 126) to open the door. If the current status of the door is unlocked, then the status of the door can change to open (line 113). The status value is sent to the `Controller` actor after any updates in all these message servers.

The `Passenger` actor is implemented to model the behavior of a passenger. We assume that the passenger can constantly send a request to the `Door` actor to open the door. This actor has only one message server (`passengerOpenDoor`). The `passengerOpenDoor` is designed to send a request (open the door) to the `Door` actor every 5 units of time (lines 125 and 128).

Timing properties. The Rebeca code in Fig. 5 contains the environment variables (denoted by `env` at the top of the code). These variables are used to set the timing parameters. The variable *networkDelayDoor* represents the amount of time that takes for a signal to get to the door from the controller (and vice versa), and the variable *networkDelayTrain* shows the amount of time that takes for a signal to get from the train to the controller (and vice versa). The other timing feature is for modeling a reaction delay of the controller when it reacts to the events (*reactionDelay*). We have *passengerPeriod* environment variable to show that the passenger can send the open command periodically (it can be modeled differently but this is the simplest way and serves our purpose to find possible errors). We also model passage of time between a train leaving and then again approaching the station (*runningTime*), and the time that train stays at the station (*atStationTime*).

The environment variables can be used as parameters to set different cycle times and communication channel features. The value for the parameters can be changed to check different configurations. For example, we can see varying depths in getting into the error state by changing the period of the passenger pressing the open door button.

5.5 Formal Verification of the Rebeca Code

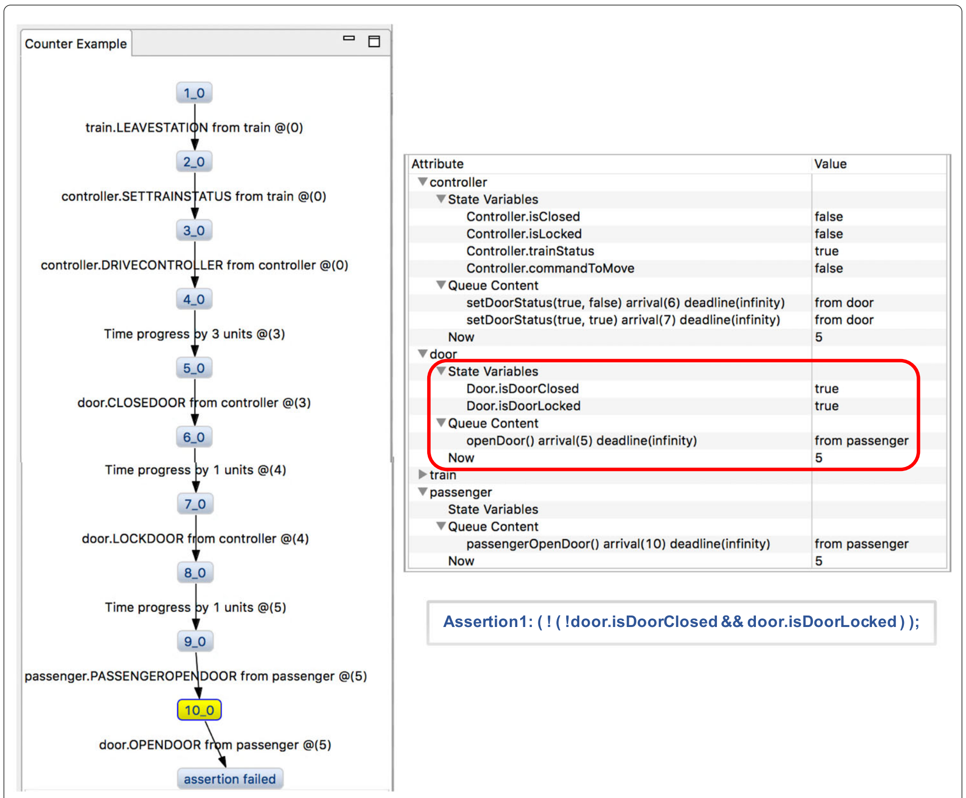
The Rebeca code in Fig. 5 is a version of the code that runs without violating any of the properties of interest. We run the Rebeca model checking tool, Afra, on a MacBook Pro laptop with 2,9 GHz Intel Core i5 processor and 8GB memory.

We check the assertion: "*It is not possible to open a locked door (not by the driver nor the passengers);*" and we show that the door cannot be opened when it is locked. This assertion covers multiple other weaker assertions, like: "*It is not possible to open a locked door (by driver or passengers) when the train is leaving the station;*", "*It is not possible to open a locked door (by driver or passengers) when the train is running;*" and "*It is not possible to open a locked door (by driver or passengers) when the train is arriving at the station*". A subset of the assertions that are checked in Afra are shown in Table 2. These assertions are written based on the state variables in the Rebeca code shown in Fig. 5, and are related to the properties explained above.

In the Rebeca code, the passenger sends a request directly to the door, the request does not pass through the controller. This is what makes the model vulnerable to errors. The door is receiving commands from both the passenger and the controller, and variant interleaving of these commands (i.e. events in the queue) may cause the execution of the model

Table 2 The properties checked by Afra in the first iteration. These assertions are satisfied for the Rebeca code shown in Fig. 5

Property
Assertion 1: (! (!door.isDoorClosed && door.isDoorLocked))
Assertion 2: (! (train.isRun && !door.isDoorLocked))
Assertion 3: (! (train.isRun && !door.isDoorClosed))



Counter Example

```
1_0
  train.LEAVESTATION from train @(0)
2_0
  controller.SETTRAINSTATUS from train @(0)
3_0
  controller.DRIVECONTROLLER from controller @(0)
4_0
  Time progress by 3 units @(3)
5_0
  door.CLOSEDOOR from controller @(3)
6_0
  Time progress by 1 units @(4)
7_0
  door.LOCKDOOR from controller @(4)
8_0
  Time progress by 1 units @(5)
9_0
  passenger.PASSENGEROPENDOOR from passenger @(5)
10_0
  door.OPENDOOR from passenger @(5)
assertion failed
```

Attribute	Value
▼controller	
▼State Variables	
Controller.isClosed	false
Controller.isLocked	false
Controller.trainStatus	true
Controller.commandToMove	false
▼Queue Content	
setDoorStatus(true, false) arrival(6) deadline(infinity)	from door
setDoorStatus(true, true) arrival(7) deadline(infinity)	from door
Now	5
▼door	
▼State Variables	
Door.isDoorClosed	true
Door.isDoorLocked	true
▼Queue Content	
openDoor() arrival(5) deadline(infinity)	from passenger
Now	5
▶train	
▼passenger	
State Variables	
▼Queue Content	
passengerOpenDoor() arrival(10) deadline(infinity)	from passenger
Now	5

Assertion1: (! (!door.isDoorClosed && door.isDoorLocked));

Fig. 6 The screenshot of Afra, coming back with a counterexample for checking the assertion "*It is not possible to open a locked door*" for a revised version of the Rebeca code in Fig. 5 where the if-statemenst in lines 102 and 112 are removed

to end in a state that violates the safety property[3]. The two "`if-statements`" in lines 102 and 112 of the reactive class `Door` are there to avoid this problem. If we remove the passenger from the model, the model is correct even without these `if-statements`.

Consider the Rebeca code in Fig. 5 where we do not have a passenger (we can just remove the statement in the main part instantiating the passenger). The number of reached states for this model is 55, and the number of reached transitions is 68 (consumed memory is 660, and the total spent time is below one second). If we have a passenger and the passenger sends a request to open the door every 5 units of time then the number of reached states will be 402079, the number of transitions is 1286068 and the total time spent for model checking is 115 seconds. If we remove the `if-statements` in lines 102 and 112, then the model violates the assertion and the model checking tool Afra comes back with a counterexample. The depth of the trace in the state space to reach the counterexample depends highly on the setting of the timing parameters.

A screenshot of the Afra tool where the counterexample is found is shown in Fig. 6. The assertion is checking the value of variables `isDoorClosed` and `isDoorLocked` from the rebec door. The screenshot shows that `isDoorClosed` is true (the door is closed), and `isDoorLocked` is also true (the door is locked). The only message in the queue of the rebec door is `openDoor` coming from passenger. This will cause the execution of the message server `openDoor` in the rebec door which will create the state in which

[3]A different design for the model, derived from a different allocation of functions in the architecture, can be modeled and model checked. More explanation will be in Section 8.

isDoorLocked stays true (the door is locked), and isDoorClosed changes to false (the door is opened). This state fails the assertion and the model checking tool comes back with the counterexample shown in Fig. 6. The counterexample states are presented on the right hand side of the figure, and the trace is in the left hand side of the figure.

Note that changing the timing parameters can change the state space significantly. The timing parameter includes the period of sending the requests, network delay, and the computation/process delay (a detailed example is described in Section 6.2).

6 The Iterative Process and Incremental Extensions: Updating and Fixing

Throughout the paper we explained one iteration of the VDD-CPS process from the requirements to Rebeca code for the door control case study. In this section we explain two more iterations. In Section 6.1 we update the Rebeca code by adding a feature that is in the requirements but not modeled, this shows how more complete increments are built based on the requirements. Section 6.2 shows how by using the VDD-CPS process we can discover a new requirement that is added to the set of requirements since it concerns concurrency issues, that is our main focus. Note that the Rebeca model in Fig. 5 is already the next increment of what is explained in the conference paper [19] where we added the "*running*" state to the code to make the code more faithful to the requirements.

6.1 Second Iteration in the VDD-CPS Process

In the second iteration, we add the concept of "*platform*" defined in the safety requirements SafeReq4 and SafeReq5 in Table 1 to the code. In the railway domain, a platform can be defined as "*an area alongside a railway track providing convenient access to trains*" [36]. This implies that passengers get on and off the train through the doors that are on the side of the train close to the platform. This is also done for safety reasons. The safety requirements SafeReq4 and SafeReq5 highlight that only the external doors that are on the side close to the platform shall be opened to prevent passengers from falling down out of the platform. By modelling the concept of "*platform*", it is possible to formally verify that the scenario in which a passenger opens an external door on the wrong side of the train does not happen.

The state diagrams and sequence diagrams given in Figs. 3 and 4 stay valid for this iteration. In order to add the functionality related to the platforms, we apply the following changes to the Rebeca code presented in Fig. 5. Instead of only one door, we have door1 and door2 instantiated from the Door reactive class. Each door has an id representing the platform close to it.

While executing the approachStation method, the train actor sets the platform id using a nondeterminitic assignment. The nondeterminitic assignment platformId = ?(1,2) models possible different behaviors. The platform id is sent to the controller actor by the train actor together with other state variables after any updates.

The Passenger actor can constantly send a request to the Door actor to open door1 or door2. The passengerOpenDoor is designed to send a request (open the door) to the Door actor every 5 units of time. Figure 7a shows the updated passengerOpenDoor message server.

As explained in Section 5.4, the setDoorStatus in controller actor is called by the Door after updating the status of the doors. Figure 7b shows the updated message

```
msgsrv passengerOpenDoor(byte doorlD){
    if (doorlD == 1){
      door1.openDoor();
      // open either the door with id 1 or the door with id 2
      self.passengerOpenDoor(?(1,2))
      after(passengerPeriod);
    }
    else if (doorlD == 2){
      door2.openDoor();
      self.passengerOpenDoor(?(1,2))
      after(passengerPeriod);
     }
}
```

(a) The updated **PassengerOpenDoor** message server from **Passenger** actor.

```
msgsrv setDoorStatus(byte doorlD, boolean close, boolean lock) {
   doorlsClosed[doorlD-1] = close;
   doorlsLocked[doorlD-1] = lock;
   if ((isClosed != (doorIsClosed[0] && doorIsClosed[1])) ||
                    (isLocked != (doorIsLocked[0] && doorlsLocked[1]))){
        isClosed = (doorIsClosed[0] && doorIsClosed[1]);
        isLocked = (doorIsLocked[0] && doorIsLocked[1]);
        self.driveController();
    }
}
```

(b) The updated **setDoorStatus** message server from **Controller** actor.

Fig. 7 The updated two message servers of the Rebeca code presented in Fig. 5

server in this iteration. We consider the `isClosed` and `isLocked` variables to show the status of both `door1` and `door2`. If both doors are closed then the value of `isClosed` is `true`. Similarly, if both doors are locked then the value of `isLocked` is `true` otherwise they are `false`.

For the Rebeca code in Fig. 5, when we have a passenger and the passenger sends a request to open the door every 5 units of time, the number of reached states is 917 and the number of reached transitions is 1235 (the total spent time is two seconds and consumed memory is 18340).

The updated code assures that a locked door on both platforms cannot be opened not only when the train is running but also when the train is at station. In particular, the doors that are on the side of the train opposite to the platform shall be kept locked. Thus, we check whether the behavioral model that is updated based on the requirements (`SafeReq4` and `SafeReq5`) violates a safety property of the train.

This also means to show that the requirements may be incorrect, inconsistent, or ambiguous.

We check the assertion: "*It is not possible to open a locked door on the opposite side of the platform;*" and we show that the door cannot be opened on the opposite side of the platform when it is locked. This assertion covers multiple other weaker assertions, i.e., "*It is not possible to open a locked door on the opposite side of the platform when the train is leaving the station;*", "*It is not possible to open a locked door on the opposite side of the platform when the train is running;*" and "*It is not possible to open a locked door on the*

Table 3 The properties checked by Afra in the second iteration. These assertions are satisfied

Property
Assertion 1: (! ((!door1.isDoorClosed && door1.isDoorLocked) || (!door2.isDoorClosed && door2.isDoorLocked)))
Assertion 2: (! ((!door1.isDoorClosed || !door2.isDoorClosed) && (door1.isDoorLocked && door2.isDoorLocked)))
Assertion 3: (! (train.isRun && (!door1.isDoorClosed || !door2.isDoorClosed)))
Assertion 4: (! (train.isRun && (!door1.isDoorLocked || !door2.isDoorLocked)))
Assertion 5: (! (platform == 1 && !door2.isDoorClosed))
Assertion 6: (! (platform == 2 && !door1.isDoorClosed))
Assertion 7: (! (platform == 1 && !door2.isDoorLocked))
Assertion 8: (! (platform == 2 && !door1.isDoorLocked))

opposite side of the platform when the train is arriving to the station". For what concerns model checking, in our experiments these properties are satisfied, confirming that the models with the given configurations did not violate the requirements. Table 3 shows some of the assertions that are checked using Afra in this iteration, these assertions are written based on the state variables in the Rebeca code, and are related to the properties explained above.

6.2 Third Iteration in the VDD-CPS Process

In the third iteration, we focus on the concurrency and timing problems to highlight the benefits of using a verification-driven approach based on Rebeca. Specifically, we are interested to verify that a shared resource, such as the external train doors in our use case, can never behave in an undesirable way due to inconsistent requests that may arrive simultaneously.

So, we iterate the SARE approach to search for new safety requirements that may be necessary to mitigate a possible failure of the lock mechanism and, consequently, avoid or reduce the probability of the hazard "*Passengers fall out of the train*".

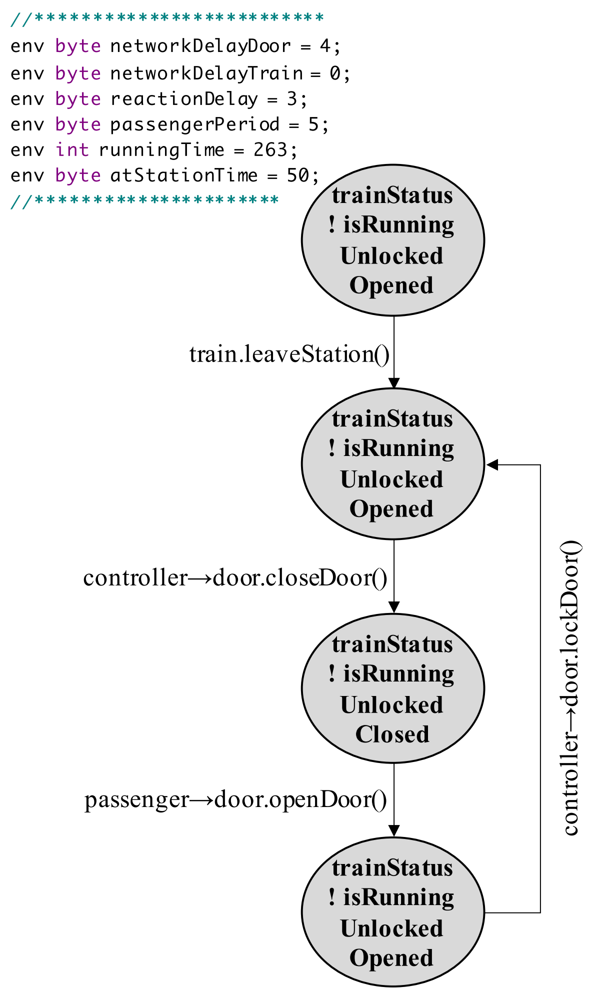
This results in a new safety requirement, i.e. `SafeReq7` in Table 4, that aims at avoiding that a passenger can open a closed door when the train is leaving the station. The lock mechanism can fail because it is "*susceptible to malfunctions*". Malfunctions can be erroneous and/or delayed inputs, inconsistent inputs, computational errors, and so on. In particular, the requirement `SafeReq7` concerns the safety behavior of the system in case a closed door receives simultaneously two or more inconsistent requests, i.e. the *open* request from a passenger and the *lock* request from the driver. The pre-condition "*the train is leaving the station*" guarantees therefore that the request to open a closed door is not performed when the train is departing, which is a safety behavior.

In this iteration, we aim at formally verifying the consequences of the interference between the two events of *open* triggered by the passenger and *lock* triggered by the controller after the doors are closed and the train is ready to leave. As explained earlier, at the beginning of the Rebeca code, we can define environment variables as parameters to

Table 4 The new safety requirement `SafeReq7` to prevent from opening an external door when the train is leaving the station

Name	Safety Requirement
SafeReq7	GIVEN the train is leaving the station AND an external door is closed WHEN a passenger requests to open an external door THEN the door controller shall keep the external door closed

set different cycle times and communication channel features. The values of these variables can be changed to check different configurations. We show that the verified Rebeca code can get into an error state by changing the settings. We set the time duration that takes for a signal to get from the door to the controller to 4 (*networkDelayDoor* = 4) and set the time duration that takes for a signal to get from the train to the controller to 0 (*networkDelayTrain* = 0). Figure 8 shows the simplified state diagram for a livelock bug when the train wants to leave the station (door should close and then lock) and a passenger wants to open the door by pressing the open button every 5 units of time. The train never reaches the *running* state. This scenario shows that the door can be closed by the controller, and opened by the passenger iteratively, resulting in blocking the train from moving. So, model checking shows that the door behaves in an undesirable and unexpected way in case of simultaneous inconsistent requests. This result corroborates the fact that a new requirement is needed to avoid this situation, such as the requirement `SafeReq7`. We also came up with other settings for the timing parameters in which the train was delayed but eventually could move and go into the *running* state.



```
//*************************
env byte networkDelayDoor = 4;
env byte networkDelayTrain = 0;
env byte reactionDelay = 3;
env byte passengerPeriod = 5;
env int runningTime = 263;
env byte atStationTime = 50;
//********************
```

trainStatus ! isRunning Unlocked Opened
train.leaveStation()
trainStatus ! isRunning Unlocked Opened
controller→door.closeDoor()
trainStatus ! isRunning Unlocked Closed
passenger→door.openDoor()
trainStatus ! isRunning Unlocked Opened
controller→door.lockDoor()

Fig. 8 The simplified state diagram of a livelock bug in the door controller case study that happens with a specific setting for timing parameters

This iteration shows how the Rebeca code can be used to check different settings for the timing parameters. This feature can be used in investigating and setting the timing parameters for the network and computation delays, and the cycle of the periodic events.

7 Related Work

Model-Driven Development (MDD) is intended to reduce complexity in the classical development approaches. Using MDD, different objectives with regard to design, verification, simulation, and code generation can be reached at different stages of development [37, 38].

MDD based on co-modelling of hardware and software is a main approach for developing cyber physical systems (CPS) that involves a combination of different computation models and communication patterns along with physical dynamics [1, 3, 39]. With the growing size and complexity of CPS, there is a need for (semi-) formal approaches to design and model the system at different stages of the development process. Currently, modelling language standards like SysML [40], MARTE [41] and MATLAB/Simulink are used by engineers in practice in some areas for modelling CPS.

After behavioral modelling of the system, reasoning on the correctness of the system behavior is the essential next phase during the development process. In general, a big part of resources during the CPS development phases are allocated to ensure that the system fulfills the requirements [42]. Verification and validation can be done using testing, simulation, and formal verification. However, with the growing complexity of CPS there are many challenges in verification and validation phases [43].

Testing approaches are often intended to generate test cases based on the internal structure of the model to evaluate different paths of execution [44–46] or act in a black-box fashion such as falsification-based techniques [47] and differential testing techniques [48] to generate the test cases resulting in violation of system requirements. However, testing is not effective enough when dealing with concurrent systems due to the non-deterministic interleaving between the processes running on distributed components. Also, testing may not be optimal and comprehensive for checking timed behavior particularly in cyber-physical systems. This issue becomes more serious for safety-critical CPS where any failure, bug or undesired situation might cause catastrophic consequences. Therefore, using formal verification for reasoning about the behavior of the system, finding bugs and undesired situations becomes more critical.

Simulation approaches, specifically the ones targeting the co-modelling of hardware and software are another underlying part of MDD chain for visualizing and also behavior verification and validation step. The maturity is growing in this domain and there are several commercial and academic tools for co-modelling and simulation of hardware and software. Ptolemy II [3] and Stateflow [49] are popular examples of this category. However, they do not support formal verification.

Formal verification and more specifically model checking is one of the main techniques for verifying different types of safety and liveness properties in safety-critical systems. Timing properties are an intrinsic aspect of CPSs. There are model checking tools which are able to capture timing features such as RMC (Rebeca Model checker) [50], UPPAAL [8], and PRISM [51]. They support different types of models such as timed automata, and timed actors.

The main challenge in using model checking tools is the state space explosion problem, another certain challenge in using formal methods within MDD chain is the mapping high-level requirements onto the formal specifications. There are several different approaches as transformation engines for addressing this challenge. In particular, there is a considerable amount of literature on transforming SysML/UML specification to inputs for different formal verification tools [52–54]. The FTG+PM framework [55, 56] is an example of such frameworks which presents formalism transformation between models within model-driven development. The framework consists of two sub languages: the Formalism Transformation Graph (FTG) and Process Model (PM) languages. The former (FTG) presents a set of available modeling languages within a given domain and the latter (PM) describes the control flow between the model transformation activities during the development life cycle. It supports automatic model transformation between different phases of design, verification, simulation, deployment and code generation. It also presumes manual transformation of textual requirements to a SysML requirement diagram in the process. Gamma [57] is another modeling framework which integrates heterogeneous statechart components to make a hierarchical composition, supports formal verification using UPPAAL for the composite model and provides automatic code generation on top of the existing source code of the components. Gamma focuses on building hierarchical statechart network based on the existing statechart components, and as the most existing tools and approaches does not support mapping of requirements onto behavioral models. In [58] an MDD framework is proposed for dataflow applications on multi-processor platforms. The framework uses Synchronous Dataflow (SDF) graphs to model application and besides the SDF, a platform application model (PAM) showing hardware platform and an allocation model are also created. The SDF model and hardware models are then transformed to priced time automata which are used as inputs to UPPAAL for verification of requirements and also to compute the energy-optimal schedule for given requirements. However, some of these solutions are ’heavy-weight’, as discussed in [59, 60], and thus not suitable as part of an iterative development approach as is one of the important aspects of our work here.

Placing our approach among others.

In modeling and analysis, the faithfulness of the model to the target system is of importance and could effectively facilitate the process [5]. The Rebeca language helps in assuring the faithfulness of the model by decreasing the semantic gap between the model and the system. Actor model is a reference model for modeling the behavior of distributed reactive systems, and also suggested for co-modeling of hardware and software of cyber-physical systems [1]. The actors in the design step are corresponding to the components in the requirements and architecture. This feature makes the transformation step less costly. Using Rebeca for modeling and verification we bridge the gap between the design models and formal verification. In this work, we use *GIVEN-WHEN-THEN* syntax to specify the safety requirements, and to fill the gap between the actor model and the requirements we use common behavioral models, i.e., UML state diagrams or sequence diagrams that are closer to the requirement specification and quite common in the industrial application domains.

8 Discussion and Future Work

In summary, our proposed process is a light-weight verification-driven iterative process for model-driven development of safety-critical cyber-physical systems. Using formal verification within the proposed MDD process makes it well suited for safety-critical domains where a solid verification of all properties is required. It involves actor-based modelling and formal verification using Timed Rebeca and the associated model checking tool Afra. Actor-based style of modelling, mitigates the issue of transformation from high-level specification to the inputs of a formal verification tool on one hand, and to an executable code in CPS domain on the other hand. Moreover, to bridge the remaining gap between high-level requirements and actor model, we leverage a structuring method based on *GIVEN-WHEN-THEN* syntax to alleviate the ambiguity and facilitate the transition from requirements to the formal model. The structured requirements also help in one of the most challenging tasks in model checking which is deriving the required properties to check.

Discussion. To reach the Rebeca code from the requirements, we use an iterative approach. There may be ambiguity in the informally stated requirements that need to be clarified. To come up with the right state variables and right transitions among states, we may need to go back and forth several times and ask the experts for the right information to avoid misunderstandings and incorrect outcome. As stated in many classical papers on formal methods, one of the main advantages of formal methods is to make the requirements clear, unambiguous, and consistent. Some examples of this kind of clarifications within our work are explained throughout the paper.

Rebeca codes can be useful for checking safety and timing properties only if the topology of the actor model matches (or is consistent with) the architecture of the system. As we plan for a straightforward mapping of Rebeca code to executable code, we need this consistency. This can be another challenge in the process, to know the architecture and the allocation of tasks to different components. One example is the decision that we made for the Door Control Unit, modeled within the actor door, to send the open command to the door upon receiving the request from the passenger. Alternatively, we could have a model in which all the decisions for sending the open command to the door are handled centrally in the Train Control Unit. This would change the design and verification results in a significant way.

In the current Rebeca code, the status of the units are sent to the control unit upon any change. Another design is updating the status of different units periodically. This will result in a complicated design where verification can help in finding the timing problems and tuning the timing features. Again, the decision has to be based on the architecture and execution model of the system.

Future work. This work serves as a foundation towards several other interesting directions. One direction to go is to make the mappings automatic or semi-automatic. The transformations among state diagrams together with sequence diagrams to Rebeca code, and generating Lingua Franca code from Timed Rebeca can be automated.

Abbreviations

MBV: Model-Based Verification; MDD: Model-Driven Development; CPS:Cyber-Physical Systems; SARE: Safety Requirements Elicitation; I/O: Input-Output; TCU: Train Control Unit; DCU: Door Control Unit; INIT: Initiating; ENV: Environment; TCTL: Timed Computational Tree Logic; VDD-CPS: Verification-Driven Development of Cyber-Physical Systems; RMC: Rebeca Model checker; SDF: Synchronous Dataflow; PAM: Platform Application Model

Acknowledgment
We would like to thank Edward Lee for reading the paper and giving us very useful comments.

Authors' contributions
The first four authors contributed equally to the original idea, proposed technique, and writing of the paper. The fifth author helped with the overall quality of the paper, formulating some of the research questions, and preparing the paper for final submissions. All authors read and approved the final manuscript.

Authors' information
Dr. Marjan Sirjani is a professor of software engineering at Mälardalen University, Sweden. Dr. Luciana Provenzano is a senior lecturer at Mälardalen University, Sweden. Dr. Sara Abbaspour Asadollah is a researcher at Mälardalen University, Sweden. Mahshid Helali Moghadam is a researcher at RISE Research Institutes of Sweden and a doctoral student at Mälardalen University, Sweden. Dr. Mehrdad Saadatmand is a senior researcher and group manager at RISE Research Institutes of Sweden.

Funding
The research of the first three authors for this work is supported by the Serendipity project funded by the Swedish Foundation for Strategic Research (SSF). The research of the first two authors is also supported by the DPAC project funded by the Knowledge Foundation (KK-stiftelsen). The research of the first author is supported by the KKS SACSys project too. The research of the fourth and fifth authors is funded partially by Vinnova through the ITEA3 European TESTOMAT and XIVT [61] projects. Open access funding provided by Mälardalen University.

Availability of data and materials
The details of the models and codes are already included in this paper. However, the Rebeca codes can also be provided upon request from the corresponding author.

Declarations

Competing interests
The authors declare that they have no competing interests.

Author details
[1]Mälardalen University, Västerås, Sweden. [2]Reykjavik University, Reykjavik, Iceland. [3]RISE Research Institutes of Sweden, Västerås, Sweden.

Received: 21 October 2019 Accepted: 3 March 2021
Published online: 24 May 2021

References
1. Lee EA. Cyber physical systems: Design challenges. In: 2008 11th IEEE International Symposium on Object and Component-oriented Real-time Distributed Computing (ISORC). IEEE; 2008. p. 363–9.
2. Lee EA. The problem with threads. Computer. 2006;39(5):33–42.
3. Ptolemaeus C. System Design, Modeling, and Simulation Using Ptolemy II, vol. 1. Berkeley: Ptolemy.org; 2014.
4. Lee EA, Sirjani M. What good are models? In: Formal Aspects of Component Software - 15th International Conference, FACS 2018, Pohang, South Korea, October 10-12, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11222. Springer; 2018. p. 3–31.
5. Sirjani M. Power is overrated, go for friendliness! expressiveness, faithfulness, and usability in modeling: the actor experience. In: Principles of Modeling. Cham: Springer; 2018. p. 423–48.
6. Holzmann G. Spin Model Checker, the: Primer and Reference Manual. 2003.
7. Cimatti A, Clarke E, Giunchiglia F, Roveri M. Nusmv: a new symbolic model checker. International Journal on Software Tools for Technology Transfer. 2000;2(4):410–25.
8. Larsen KG, Pettersson P, Yi W. Uppaal in a nutshell. Int J Softw Tools Technol Tran (STTT). 1997;1(1):134–52.
9. Reynisson AH, Sirjani M, Aceto L, Cimini M, Jafari A, Ingólfsdóttir A, Sigurdarson SH. Modelling and simulation of asynchronous real-time systems using timed Rebeca. Sci Comput Program. 2014;89:41–68.
10. Sirjani M, Khamespanah E. On time actors. In: Theory and Practice of Formal Methods. Cham: Springer; 2016. p. 373–392.
11. Khamespanah E, Sirjani M, Sabahi-Kaviani Z, Khosravi R, Izadi M. Timed Rebeca schedulability and deadlock freedom analysis using bounded floating time transition system. Sci Comput Program. 2015;98:184–204.
12. Sirjani M, Lee EA, Khamespanah E. Verification of cyberphysical systems. Mathematics. 2020;8(7):1068.
13. Zarneshan M, Ghassemi F, Sirjani M. Formal modeling and analysis of medical systems. In: International Conference on Coordination Languages and Models. Cham: Springer; 2020. p. 386–402.
14. Moradi F, Asadollah SA, Sedaghatbaf A, Čaušević A, Sirjani M, Talcott C. An actor-based approach for security analysis of cyber-physical systems. In: International Conference on Formal Methods for Industrial Critical Systems. Cham: Springer; 2020. p. 130–47.
15. Rebeca. Rebeca Homepage. http://www.rebeca-lang.org/. Retrieved July 2019.
16. Sirjani M, Movaghar A, Shali A, De Boer FS. Modeling and verification of reactive systems using Rebeca. Fundam Informaticae. 2004;63(4):385–410.
17. Sirjani M. Rebeca: Theory, applications, and tools. In: International Symposium on Formal Methods for Components and Objects. Berlin: Springer; 2006. p. 102–26.

18. de Boer FS, Serbanescu V, Hähnle R, Henrio L, Rochas J, Din CC, Johnsen EB, Sirjani M, Khamespanah E, Fernandez-Reyes K, Yang AM. A survey of active object languages. ACM Comput Surv. 2017;50(5):76–17639.
19. Sirjani M, Provenzano L, Asadollah SA, Moghadam MH. From requirements to verifiable executable models using Rebeca. In: International Workshop on Automated and Verifiable Software sYstem DEvelopment; 2019. http://www.es.mdh.se/publications/5645-.
20. Provenzano L, Hänninen K, Zhou J, Lundqvist K. An ontological approach to elicit safety requirements. In: 2017 24th Asia-Pacific Software Engineering Conference (APSEC). IEEE; 2017. p. 713–8.
21. Firesmith DG. A taxonomy of safety-related requirements. Requir Eng. 2004. White paper - Carnegie Mellon University.
22. Zhou J, Hänninen K, Lundqvist K, Provenzano L. An ontological approach to hazard identification for safety-critical systems. In: 2017 Second International Conference on Reliability Systems Engineering (ICRSE). IEEE; 2017. p. 1–7.
23. Zhou J, Hänninen K, Lundqvist K, Provenzano L. An ontological approach to identify the causes of hazards for safety-critical systems. In: 2017 2nd International Conference on System Reliability and Safety (ICSRS). IEEE; 2017. p. 405–13.
24. Fowler M. ThoughtWorks: GivenWhenThen. 2013. https://martinfowler.com/bliki/GivenWhenThen.html. Retrieved July 2019.
25. North D. Introducing BDD. Better Softw Mag. 2006. https://dannorth.net/introducing-bdd/. Retrieved July 2019.
26. Agha GA. ACTORS - a Model of Concurrent Computation in Distributed Systems. MIT Press series in artificial intelligence. Cambridge: MIT Press; 1990.
27. Hewitt C. Description and theoretical analysis (using schemata) of PLANNER: A language for proving theorems and manipulating models in a robot. Technical report, MIT Artificial Intelligence Technical Report. 1972.
28. Khamespanah E, Sirjani M, Mechitov K, Agha G. Modeling and analyzing real-time wireless sensor and actuator networks using actors and model checking. STTT. 2018;20(5):547–61.
29. Yousefi B, Ghassemi F, Khosravi R. Modeling and efficient verification of wireless ad hoc networks. Form Asp Comput. 2017;29(6):1051–86.
30. Sharifi Z, Mosaffa M, Mohammadi S, Sirjani M. Functional and performance analysis of network-on-chips using actor-based modeling and formal verification. ECEASST. 2013;66:1–16.
31. Nigro L, Sciammarella PF. Time synchronization in wireless sensor networks: A modeling and analysis experience using theatre. In: 2018 IEEE/ACM 22nd International Symposium on Distributed Simulation and Real Time Applications (DS-RT). IEEE; 2018. p. 1–8.
32. Lohstroh M, Lee EA. Deterministic actors. In: 2019 Forum for Specification and Design Languages (FDL). Southampton: IEEE; 2019. p. 1–8.
33. Lohstroh M, Schoeberl M, Goens A, Wasicek A, Gill C, Sirjani M, Lee EA. Actors revisited for time-critical systems. In: Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019. Las Vegas: IEEE; 2019. p. 1–4.
34. Rebeca. Afra Tool. 2019. http://rebeca-lang.org/alltools/Afra. Retrieved July 2019.
35. Rebeca. RMC Tool. 2016. http://rebeca-lang.org/alltools/RMC. Retrieved July 2019.
36. Wikipedia. RailwayPlatform. 2019. https://en.wikipedia.org/wiki/Railway_platform. Retrieved July 2019.
37. Beydeda S, Book M, Gruhn V, et al. Model-driven Software Development. Heidelberg: Springer; 2005.
38. Liebel G, Marko N, Tichy M, Leitner A, Hansson J. Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice. Softw Syst Model. 2018;17(1):91–113.
39. Lee EA. CPS Foundations. In: Proceedings of the 47th Design Automation Conference (DAC '10). New York: Association for Computing Machinery; 2010. p. 737–42.
40. Object Management Group. OMG Systems Modeling Language v1.5. 2017. https://sysmlforum.com/sysml-specs/. Retrieved July 2019.
41. Object Management Group. UML profile for MARTE, beta 2. 2008. https://www.omg.org/omgmarte/Specification.htm. Retrieved July 2019.
42. Baheti R, Gill H. Cyber-physical systems. Impact Control Technol. 2011;12(1):161–6.
43. Chaturvedi DK. Modeling and Simulation of Systems Using MATLAB and Simulink. Boca Raton: CRC press; 2017.
44. Nejati S. Testing cyber-physical systems via evolutionary algorithms and machine learning. In: Proceedings of the 12th International Workshop on Search-Based Software Testing (SBST '19). Montreal: IEEE Press; 2019. p. 1.
45. Matinnejad R, Nejati S, Briand L, Bruckmann T, Poull C. Automated model-in-the-loop testing of continuous controllers using search. In: Ruhe G, Zhang Y, editors. Search Based Software Engineering. Berlin, Heidelberg: Springer; 2013. p. 141–57.
46. Marinescu R, Saadatmand M, Bucaioni A, Seceleanu C, Pettersson P. A model-based testing framework for automotive embedded systems. In: 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications. IEEE; 2014. p. 38–47.
47. Kong S, Gao S, Chen W, Clarke E. dreach: δ-reachability analysis for hybrid systems. In: International Conference on TOOLS and Algorithms for the Construction and Analysis of Systems. Berlin: Springer; 2015. p. 200–5.
48. Chowdhury SA, Mohian S, Mehra S, Gawsane S, Johnson TT, Csallner C. Automatically finding bugs in a commercial cyber-physical system development tool chain with slforge. In: Proceedings of the 40th International Conference on Software Engineering (ICSE '18). New York: Association for Computing Machinery; 2018. p. 981–92.
49. MathWorks. Stateflow: Model and simulate decision logic using state machines and flow charts. 2018. https://www.mathworks.com/products/stateflow.html. Retrieved July 2019.
50. Khamespanah E, Khosravi R, Sirjani M. An efficient TCTL model checking algorithm and a reduction technique for verification of timed actor models. Sci Comput Program. 2018;153:1–29.
51. Kwiatkowska M, Norman G, Parker D. PRISM: Probabilistic symbolic model checker. In: International Conference on Modelling Techniques and Tools for Computer Performance Evaluation. Berlin: Springer; 2002. p. 200–4.
52. Ando T, Yatsu H, Kong W, Hisazumi K, Fukuda A. Translation rules of SysML state machine diagrams into CSP# toward formal model checking. Int J Web Inf Syst. 2014;10(2):151–69.
53. Hansen HH, Ketema J, Luttik B, Mousavi MR, Van De Pol J. Towards model checking executable UML specifications in mCRL2. Innov Syst Softw Eng. 2010;6(1-2):83–90.

54. Andrade E, Maciel P, Callou G, Nogueira B. A methodology for mapping SysML activity diagram to time petri net for requirement validation of embedded real-time systems with energy constraints. In: 2009 Third International Conference on Digital Society. Cancun: IEEE; 2009. p. 266–71.
55. Lucio L, Mustafiz S, Denil J, Meyers B, Vangheluwe H. The formalism transformation graph as a guide to model driven engineering. McGill University, Tech. Rep. SOCS-TR2012. 2012.
56. Lucio L, Denil J, Vangheluwe H. An overview of model transformations for a simple automotive power window. Technical report, McGill University, Tech. Rep. SOCS-TR-2012.2, 2012, http://msdl.cs.mcgill~.... 2012.
57. Molnár V, Graics B, Vörös A, Majzik I, Varró D. The gamma statechart composition framework: Design, verification and code generation for component-based reactive systems. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion). IEEE; 2018. p. 113–6.
58. Ahmad W, Yildiz BM, Rensink A, Stoelinga M. A model-driven framework for hardware-software co-design of dataflow applications. In: International Workshop on Design, Modeling, and Evaluation of Cyber Physical Systems. Pittsburgh: Springer; 2016. p. 1–16.
59. Gilliam DP, Powell JD, Bishop M. Application of lightweight formal methods to software security. In: 14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise (WETICE'05); 2005. p. 160–5.
60. Souaf S, Loulergue F. A first step in the translation of alloy to coq. In: Ait-Ameur Y, Qin S, editors. Formal Methods and Software Engineering. Cham: Springer; 2019. p. 455–69.
61. Schlingloff H, Kruse PM, Saadatmand M. Excellence in variant testing. In: Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems; 2020. p. 1–2.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Submit your manuscript to a SpringerOpen journal and benefit from:

► Convenient online submission
► Rigorous peer review
► Open access: articles freely available online
► High visibility within the field
► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com