Approaches to Cyber-physical (model) Model-checking

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Dipl.-Ing. Christoph Luckeneder
Registration Number e0927141

to the Faculty of Electrical Engineering and Information Technology
at the TU Wien

Advisor: Univ.Prof. Dr.techn. Hermann Kaindl
Second advisor: Dipl.-Ing. Dr.techn. Ralph Hoch, BSc

The dissertation has been reviewed by:
Univ.Prof. Dr. Óscar Pastor
Universitat Politècnica de València, Spain.

Univ.Prof. Dr. Xavier Franch
Universitat Politècnica de Catalunya, Spain.

Declaration of Authorship

Dipl.-Ing. Christoph Luckeneder

I hereby declare that I have written this Doctoral Thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

Vienna, 14th November, 2022

Christoph Luckeneder

Acknowledgments

First, I would like to thank my parents, who supported me morally and financially. Without their help moving to Vienna and starting my academic journey would not have been possible.

I also want to thank my girlfriend. Without her, I would not have had the guts to go to Vienna. She also helped me with warm words and deeds when I lost hope along the journey. I am looking forward to many more great years together.

Lastly, I would like to thank my advisor, Prof. Hermann Kaindl, and my assistant advisor and former co-researcher, Dr. Ralph Hoch. During our conversations and sometimes disagreements over details, I always found new ways to look at topics. They were always open to new ideas and a source of inspiration for me as I worked on my thesis.

Part of this research has been carried out in the FeatureOpt project (No. 849928), funded by the Austrian BMVIT (represented by the Austrian FFG), the VerASoS project (No. 861210), funded by the Austrian FFG, and the project ForSAMARA, which is nominated for cascaded funding under the H2020 project RobMoSys (No.732410).

Abstract

Formal verification of complex systems like cyber-physical systems (CPS) is a major challenge. The reason is, among other things, state space explosion, i.e., the phenomenon that the number of possible states is exponential to the number of variables used to specify the system for the purpose of verification through model-checking. In addition to software models to be verified, for CPS verification models of the physical environment have to be included as well, but they are significantly different from software models. This entails further problems with CPS verification through model-checking.

Hence, the main challenge addressed in this thesis has been the creation of cyber-physical models for model-checking. In particular, it was challenging to provide models that contain all the necessary information for proving or disproving a given property and still have a reasonable size for efficient verification. More specifically, this thesis dealt with three research questions that are all related to this challenge. However, each of them emphasizes a different aspect and was dealt with in a different domain.

The first research question asks whether and how formal verification may help to improve models in a top-down design process of CPS. During the design of a CPS, often the challenge arises of verifying a system model against certain properties. In an early design stage, however, the system model may not include sufficient detail. The physical environment is often described initially by formulas, like Newton's law. In this thesis, a top-down design approach is proposed and evaluated (in the automotive domain) that starts with an abstract (qualitative) model of the system – including the physical parts – and refines it systematically while including both verification and validation.

The second research question asks how to formally check consistency between process-oriented models and models of the environment as early as possible in the course of CPS development. This thesis presents an approach to verifying consistency between UML Activity diagrams modeling processes, object life cycles modeling the environment, and context-dependent semantic action specifications. It is evaluated in the domain of charging electric vehicles.

The third research question asks how the runtime efficiency of the formal verification of certain robot applications using structural abstractions based on a voxel representation can be improved. Due to the complexity of robot applications and their complex work environment modeled as a voxel grid, the formal verification of such applications is usually time-consuming. This thesis presents and evaluates an approach that uses selective refinement of structural abstractions to improve the runtime efficiency of model-checking.

Summarizing, this thesis addresses certain limitations of model-checking CPS models and proposes new approaches for improvements especially with regard to models of the environment.

Contents

CHAPTER 1

Introduction

Unlike testing and simulation, model-checking can formally prove that a system conforms to a specific property. Model-checking is used in a wide variety of domains, but it is especially important when dealing with safety-critical systems. For applying model-checking to such systems, a formalization of the real-world system through models is necessary. Hence, model-checking is performed on abstract models and not the implementation itself.

A particularly interesting domain is systems involving a real-world environment and a computer part, i.e., a cyber-physical system. A cyber-physical system is a system that is deeply interconnected to its environment via sensors and actuators and may also be connected to other cyber-physical systems (CPS). Because of the deep interconnection, verifying such systems is impossible without a representation of the environment. Thus, models for verification may become very complex. This poses a problem since one major limiting factor of formal verification, in particular model-checking, is the complexity of models that the verification algorithms can handle without leading to long verification times or even rendering models impossible to verify via model-checking.

To address this issue, many approaches can be viable for reducing the complexity and modeling the interactions of cyber-physical systems. In this thesis, various approaches for reducing complexity through abstraction are explored and investigated.

1.1 Problem Statement

Modeling for model-checking is a major challenge since the models must both provide the necessary information to perform the verification task and the complexity has to be manageable, to ensure efficient verification.

In this thesis, we address several issues related to creating, connecting, and transforming models of CPSs and their environment for the purpose of formal verification via model-checking.

In each of the main chapters (3-5), we address one of the following research questions:

1. How can formal verification help to improve models in a top-down design process of CPS?

   We ask how formal verification can be integrated into a top-down development process to derive concrete models from abstract ones. It is assumed that the abstract models, including the environment model(s), are defined solely based on the experience of the designer/domain expert and to only contain qualitative values of parameters and physical properties.

   They do not incorporate any concrete values of parameters and physical properties. The latter should be determined in the course of the top-down design process.

2. How to check consistency between process-oriented CPSs and their environment as early as possible in the course of the development?

   As stated above, a CPS is a system deeply interconnected with its environment. Thus, operating the same CPS in a different environment may lead to different results. We ask how models can be connected with an environment and how these models can be utilized for formal verification. We specifically focus on process-oriented CPSs with interactions with an environment and investigate how we can formally verify that all processes end in a defined state.

3. How to improve the runtime efficiency of the formal verification of certain robot applications using structural abstraction based on a voxel representation?

   This question arose while using an existing methodology for verifying robot applications. Using this methodology, complex robot applications lead to complex models – behavioral models of the robot system and models of the environment in the form of a voxel grid – and to an even more complex input model of the model-checker. Therefore, the verification becomes inefficient – in terms of verification times – for complex scenarios. The aim of this question is to investigate the potential of using abstractions of structural information (implicitly) given by the voxel grid to generate environment representations that lead to shorter verification times.

1.2 Motivation

As a research team working in the field of software-intensive systems, including cyber-physical systems (CPS), we often do projects that involve some kind of CPS.

In those projects, we found that verification of such systems is mostly done via simulation and testing. However, especially regarding safety-critical applications, we saw the need for a formal way to verify such a system to prove that the system behaves as intended. Unfortunately, we found it challenging to transfer methodologies for the formal verification of software systems to them. We also saw that very little work was done regarding the formal verification of CPSs.

1.3 Structure of the Thesis

Following this introduction, the remainder of this thesis is structured in the following manner.

While this thesis is split into three major parts, Chapter 2 provides background material on the common ground of all parts. It starts with a brief introduction to UML and especially UML Activity diagrams as well as UML States Machines, and how they can be used to model system behavior. After that, we explain verification and validation (V&V). Since in this thesis the use of model-checking is central, an introduction to the essence of model-checking is given. Finally, a brief introduction to Counter-example guided Abstraction Refinement (CEGAR) is given, a technique often used in the context of model-checking because of the inherent combinatorics.

Chapter 3 deals with verification of feature coordination through model-checking. First, specific background material on feature interaction is provided, including the reason why feature coordination is needed. After that, the running example of this chapter – Adaptive Cruise Control (ACC) – is introduced, as a combination of the two features Cruise Control (CC) and Distance Control (DC), which have to be coordinated because of their feature interaction. Then it is shown how we modeled ACC and the (simple) physics involved qualitatively using states machines. Since the verification is done on abstract high-level models and those models may not represent the real world correctly, the subsequent section presents a systematic top-down design approach with integrated model validation and verification. We explain an application of this workflow on the ACC example and present the results. This chapter concludes with related work and a comparison of our approach with the literature.

Chapter 4 first provides some background material on processes and on modeling processes. After the introduction of the running example of this chapter, the process of charging an electric car (ECar), we show how UML Activity diagrams and object life cycles (OLCs) can be used to model such processes and their interaction with each other and the environment. Then our workflow for verification of such models is shown. This section provides also some detail on how Activity diagrams are annotated with semantic information to connect them to OLCs. The next section explains in some detail the automated code-generation for model-checking that we devised. It starts with the general concept and subsequently shows how OLCs and Activity diagrams are transformed into nuXmv Code. Based on this fully implemented approach, we present a case study of applying it for iterations of verification and validation on the ECar example and the lessons learned. Also, this chapter concludes with related work and a comparison of our approach with the literature.

Chapter 5 starts with a background section describing our running example and a previously defined verification methodology for robot applications, and giving some background information on voxel grids. The following main section of this chapter first motivates why we consider structural abstraction and refinement important. In the following, we show how voxels of a voxel grid can be combined to form abstract voxels and, at the same time, guarantee over-approximation. Based on this approach for combining voxels, we then define a CEGAR-based verification workflow. The section concludes with how we integrated this workflow into the existing verification methodology for robot applications. We then apply the workflow to our running example and analyze the results. The chapter concludes with related work and a discussion.

In Chapter 6, we discuss the assumptions made and possible limitations of our approaches. We also provide ideas for future research. Chapter 7 provides our overall conclusion of this thesis.

CHAPTER 2

Background

This chapter provides background information that is needed throughout the thesis. However, since individual chapters of this thesis are partly based on more specific background each chapter additionally has its own section providing more background information.

The chapter starts by giving a short introduction to Unified Modeling Language (UML). After that, two major modeling paradigms for modeling behavior in UML, namely *Activity Diagrams* and *State Machine Diagrams*, are introduced and explained in more detail. Thereby the focus lies on the definition of model elements later used during the course of the thesis.

Subsequently, the validation and verification problem is framed by highlighting the difference between those two procedures performed during the systems engineering process.

After that, a short introduction to model-checking, a method for formal verification of behavioral models, is given. This section then focuses more on two different types of logics used to define properties to be verified and how those logics are related.

Finally, we present background information on Counter-example Guided Abstraction Refinement (CEGAR), a verification technique for verifying complex behavioral models by utilizing abstraction.

2.1 Unified Modeling Language

The Unified Modeling Language (UML)[6] is a graphical modeling language defined by the Object Management Group (OMG). It provides a standard way to model system designs and their artifacts by defining an abstract syntax and the semantics of each modeling concept. In addition, UML also defines human-readable textual and graphical notation elements for representation of the defined modeling concepts and rules how to combine those elements to form instances of different diagram types.

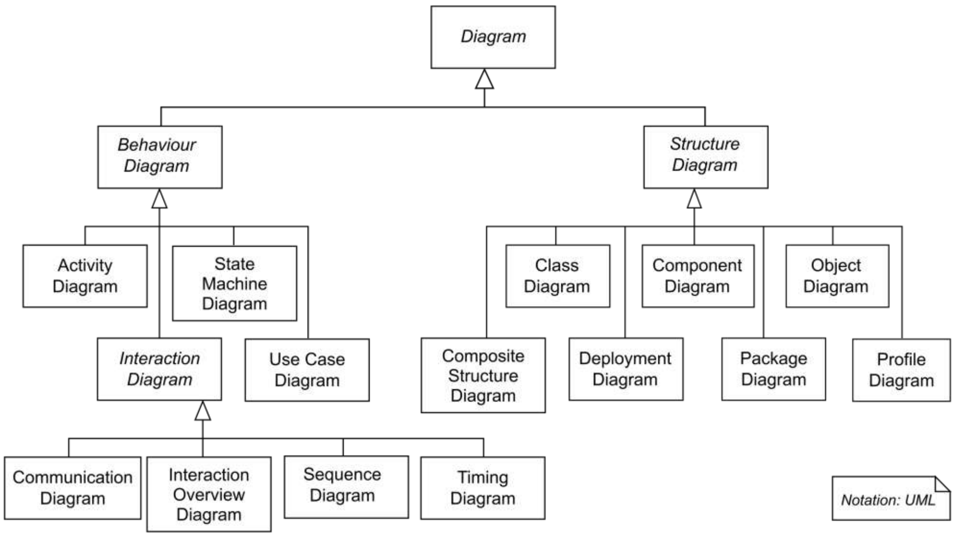


Figure 2.1: Taxonomy of UML Diagrams – Figure by Paulo Merson [61]

In total, UML defines 14 types of diagrams, each of which has a different scope. As Figure 2.1 shows there are two main types of diagrams, *Behaviour* and *Structure Diagrams*. In *Behaviour Diagrams* dynamic behavior of objects is modeled, whereas *Structure Diagrams* show static structures of the system. In this thesis, we mainly deal with diagrams of the former type, the *Activity Diagrams* and *State Machine Diagrams*. However, those diagram types are not independent from each other. Hence other types of diagrams may be used to model other aspects of the overall system.

By design, UML is a general-purpose (modeling) language. However, it provides UML Profiles as a mechanism to adapt UML to specific domains and platforms. The OMG itself uses Profiles, among other things, to define the *Systems Modeling Language* (SysML)[1] tailored to the domain of systems engineering [8] and to add functionalists for modeling of real-time (embedded) systems via the UML Profile *Modeling and Analysis of Real-Time and Embedded systems* (MARTE)[7]. In this thesis, UML is extended using a UML Profile to allow the annotation of certain model elements.

In this thesis, we use two slightly different notations. One is the notation of the tool *Eclipse* plugin *Papyrus* [10], and a generic one. The former is used when actual implementations of models are shown, and the latter is used when those models are too complex or the artifacts to be shown are scattered over multiple *Papyrus* diagrams. However, because the differences are negligible, in this section only the *Papyrus* notation is shown, unless otherwise stated.

The running example used to explain how elements are drawn together to create a whole Activity diagram is an adapted version of the “payment handling” process of [76, p. 108].

[1]Status version 1.6 [8]; SysML version 2 is planned to be specified as UML Profile and independent metamodel [5].

2.1.1 Activity Diagrams

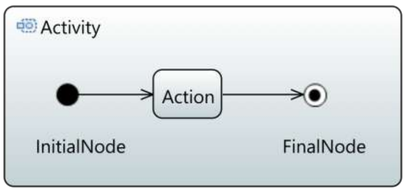
Activity Diagrams are a graphical representation of workflows. It does not matter whether the workflow is computational, e.g., an algorithm to solve a computational problem, or organizational, e.g., a business process of a company. Formally, an *Activity Diagram* is the graphical representation of an *Activity*, which itself is a way of defining behavior in UML and actually models the workflow.

An *Activity (Diagram)* mainly consists of *Activity Nodes* and *Activity Edges* connecting those nodes. The flow of execution is defined by a token system derived from Petri Nets, where the tokens can be passed from one *Activity Node* to another one through *Activity Edges*. An *Activity Node* can only be executed, if on each of its incoming edges a token is available. There are three different types of *Activity Nodes* – *Control Nodes*, *Object Nodes* and *Executable Nodes* – as well as two types of *Activity Edges* – *Control Flow* and *Object Flow*.

In the following we give more details on selected UML elements and their notation.

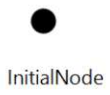
Activity:

As stated an *Activity* is the element that actually models behavior. The *Activity* is represented as the canvas. The other elements are drawn on it. In this diagram the *Activity* consists of only five elements, one *Initial Node*, one *Action*, one *Final Node* and two *Control Flows*.

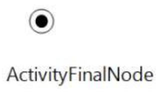


Control Nodes:

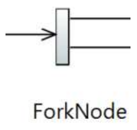
- An *Initial Node* acts as a starting point of the *Activity*.



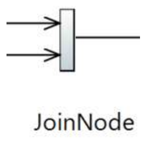
- An *Activity Final Node* stops the execution of the *Activity* if a token is received.



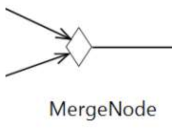
- *Fork Nodes* split flows into multiple parallel flows. To do so, a copy of the token offered on the incoming edge is offered on each outgoing edge.



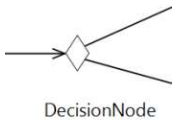
- A *Join Node* joins two or more parallel flows to one output flow. A token on the outgoing edge is only offered when on each incoming edge a token is offered.



- A *Merge Node* merges two or more flows. For each token on an incoming edge, a token on the outgoing edge is offered.

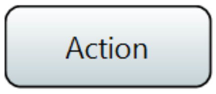


- A *Decision Nodes* is a node that routs a token on the incoming edge to one of the outgoing edges. On which outgoing edge the token is offered is determined by guard conditions defined on the outgoing edges.

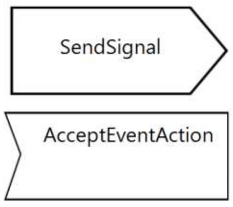


Activity Nodes and *Object Nodes*:

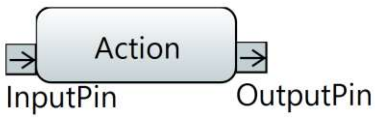
- *Actions* are *Executable Nodes* and the fundamental building blocks of activities. The execution of an *Action* represents some transformation or processing in the modeled system. However, an *Action* may also trigger the execution of a behavior, e.g., an *Activity*. In this sense an *Action* provides some sort of abstraction of more or less complex behavior. UML defines the type *Action* as generalization of many types like *Opaque Action*, *Call Behavior Action* and *Value Specification Action*[2]. The given notation stands for a wide range of different types of action. In this thesis the actions are named in such a way that the type may be inferred easily.



- *Send Signal Actions* and *Receive Event Actions* are *Actions* with special notations used to send and receive objects asynchronously. In this thesis, *Send Signal Action* and *Receive Event Actions* are used in combination to realize communication between different activities.



- *Pins* are *Object Nodes*. Depending on the subtype, *Input Pin* or *Output Pin* provide values to or accept output values from an *Action*. Pins are depicted as rectangular boxes attached to actions and their direction (input or output) is indicated by the arrows given in the corresponding box.



[2]For details, on the different types of actions, we refer to [6].

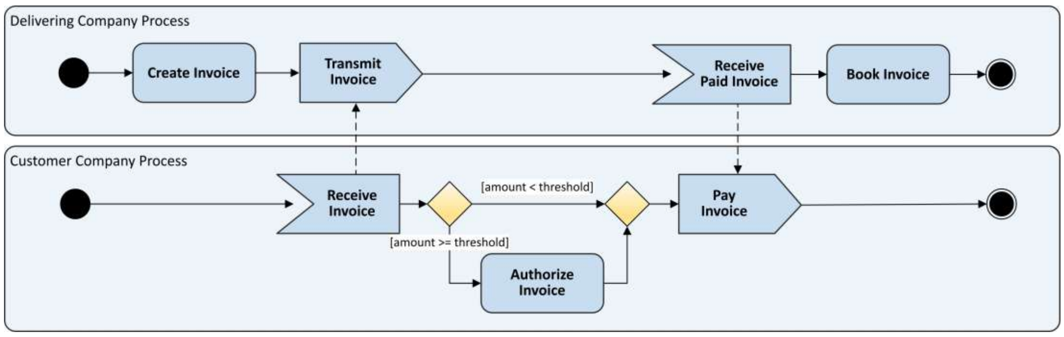
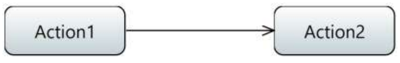


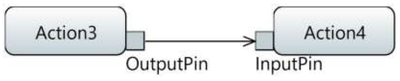
Figure 2.2: Invoice Example as Activity diagrams given in the generic notation (revised version of [47, Fig. 2]) – dashed arrows indicate which Send and Receive Actions are connected via signal.

Activity Edges:

- A *Control Flow* is an *Activity Edge* that only passes control tokens from an *Activity Node* to another one. *Control Flows* are used to explicitly model the order in which the *Activity Nodes* are executed.



- An *Object Flow* passes object tokens that may hold between *Object Nodes*. In this case, the *Object Flow* connects the *Output Pin* of *Action3* with the *Input Pin* of *Action4*. Please note that *Papyrus* changes the notation of a pin when an *Object Flows* is connected to it.



For illustrating how those elements are drawn together to form UML Activity diagrams, Figure 2.2 shows the “payment handling” process involving a delivering company and a costumer. The whole business process consists of two Activities, one for each involved party. In *Papyrus*, the two Activities are modeled in different diagrams and may even be modeled in different UML-files. However, for this representation we chose to show them side by side for a better overview.

This “payment handling” process has two (sub-)processes, the *Delivering Company Process* and the *Customer Company Process*, executed by two different parties involved in the process. As the names of the processes already hint, the former is executed by the company selling a product or providing a service, and the latter is executed by the company consuming the product or service.

Each process is started individually. However, they are synchronized. The points of synchronization are illustrated by dotted lines given in Figure 2.2. During executing the “payment handling” process those points are also considered the only points where information can be exchanged between *Delivering Company Process* and the *Customer Company Process*.

The process of the delivering company starts with creating an invoice via the *Create Invoice* action. Afterward, the *Transmit Invoice* action transmits the invoice to the customer company. After that, the *Delivering Company Process* proceeds immediately to the *Receive Paid Invoice* action, where it waits to receive the payment from the customer.

In the meantime, the customer company has also stated its process. However, it is almost immediately blocked by the *Receive Invoice* action since it is waiting for an incoming invoice. After receiving an invoice from the *Delivering Company Process*, the further advancement of the process is determined by the amount associated with the invoice. An invoice with an amount less than a certain threshold is paid immediately, whereas an invoice with an amount greater or equal to the threshold must be authorized first. Either way, the invoice is paid by executing the *Pay Invoice* action, which also sends an asynchronous notification to *Delivering Company Process*. After that, the *Customer Company Process* ends.

Receiving the notification unblocks the *Delivering Company Process*, which then proceeds which its last action *Book Invoice*, which books the invoice in the accounting.

During the thesis, two slightly different versions of the “payment handling” process are used – one with and one without the *Authorize Invoice* action. Since the additional step of authorization is more likely in big companies, we call the process containing the *Authorize Invoice* action *Big Costumer Company* process and the one without *Small Costumer Company* process.

2.1.2 State Machine Diagrams

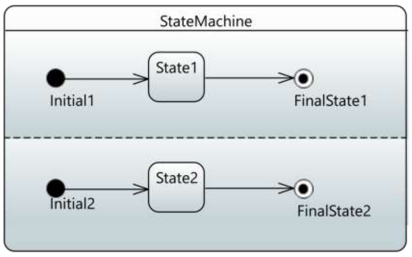
A *State Machine Diagram* is a graph that represents a *State Machine*. UML distinguishes between two types of *State Machines* – behavior state machines and protocol state machines. This thesis uses state machines to describe the behavior of CPSs or their environment and, therefore, the former.

A behavior *State Machine* comprises one or more Regions. Each *Region* contains *Vertices* and arcs (*Transitions*), connecting them, that form a graph. Events trigger the execution of *State Machines* and determine the order in which the graphs are traversed. A set of valid path traversals represents a particular execution of a *State Machine*.

During the traversal of a *State Machine*, behaviors may be executed. The order in which those behaviors are executed is determined by the order in which the graphs are traversed.

On the following page, we show the basic building blocks of *State Machines* and their notation in *Papyrus*.

State Machine:



The *State Machine* is the canvas it is drawn upon and has at least one *Region*. The diagram shown depicts a *State Machine* with two *Regions* (unfortunately, *Papyrus* does not display their names) divided by the dashed line. The behavioral flow within a *Region* is defined by a set of vertices and transitions. In our example each *Region* consists of five elements, two *Transitions* and three *Vertices* (*Initial State*, *State* and *Final State*). *Regions* are considered orthogonal to each other, if the sets of vertices and transitions are mutually exclusive and part of the same *State Machine* (or a *composite State*). If this is the case, regions are executed concurrently.

Vertices and *Transitions*:

InitialState

- An *initial Pseudostate* represents the point from which the execution of a *Region* starts. (Strictly speaking, this is only true, if the *Region* is entered via the default activation. But in this thesis all *Regions* used are activated in this way.)

- A *State* models some sort of invariant that holds during “being in this state”. In most cases this invariant is only given implicitly, e.g., by naming the *State* accordingly.

FinalState

- A *Final State* is a special kind of *State* that, if activated, signals that the *Region* has completed and no further behavior will be executed by it.

State1
State2

- UML defines a *Transition* as a directed arc from a source *Vertex* to a target *Vertex*. Source and target may be the same. A *Transition* may have an associated behavior, which is executed whenever the *Transition* is taken.

So far, we have not discussed triggers and guards associated with *Transitions*. During modeling state machines in UML, we found that the notation used for triggers and guards, displayed in *Papyrus*, is not optimal for our use case. Therefore, this thesis uses a different notation. To better understand the notation we use during the course of this thesis, we explain it using the example *State Machine* called *Example FSM*, which is given in Figure 2.3. In the top region of the state machine, the outgoing *Transition* is

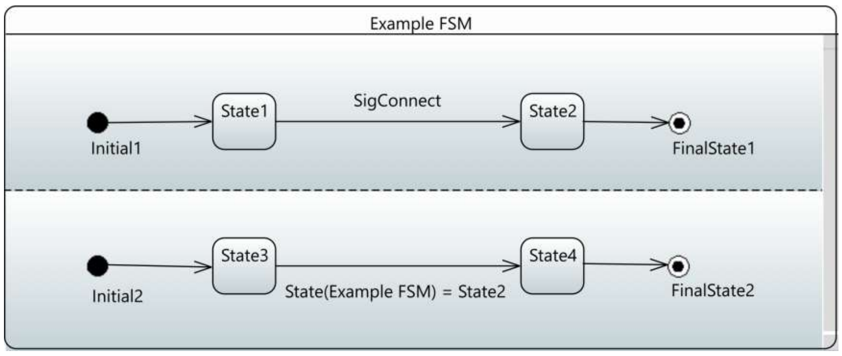


Figure 2.3: Example State Machine to explain the notations

labels with *SigConnect*. This means, given the region is currently in the *State1* the state can only change to *State2* if the signal *SigConnect* has changed its value. In the bottom region of the state machine, the label *State(Example FSM) = State2* indicates that this transition is only traversed and, therefore, the state may only change from *State3* to *State4*, if *State2* of our *Example FSM* is active.

2.2 Validation and Verification Problem

Systems verification and validation (V&V) are independent procedures that are considered critical components of a systems engineering process [3]. However, those procedures are intertwined in such a way that there are used together and complement each other in order to check if a system meets the requirements and specification and fulfills its intended purpose [3].

The following definitions are directly taken from the *IEEE Standard for System and Software Verification and Validation* and show the difference between those procedures.

Validation: "The process of providing evidence that the system, software, or hardware and its associated products satisfy requirements allocated to it at the end of each life cycle activity, solve the right problem (e.g., correctly model physical laws, implement business rules, and use the proper system assumptions), and satisfy intended use and user needs"[3]

Verification: "The process of providing objective evidence that the system, software, or hardware and its associated products conform to requirements (e.g., for correctness, completeness, consistency, and accuracy) for all life cycle activities during each life cycle process (acquisition, supply, development, operation, and maintenance); satisfy standards, practices, and conventions during life cycle processes; and successfully complete each life cycle activity and satisfy all the criteria for initiating succeeding life cycle activities. Verification of interim work products is essential for proper understanding and assessment of the life cycle phase product(s)."[3]

In short, *validation* provides evidence if the system meets the needs of the customer and other stakeholders and *verification* provides evidence if the system was built right/in the right way [4].

The validation and verification problem arises from the fact that verification alone does not provide evidence that fulfills the intended purpose. To complicate things, this thesis deals with physical systems or systems embedded into a physical environment. In this context, we consider validation to have two roles, checking whether the system fulfills the intended purpose and if the physical part of the system and the environment were modeled realistically.

Although in this thesis we mainly deal with model-checking, which is considered a tool for formal *verification*, also *validation* plays an imported role. This is because a model-checker alone can only state whether a model fulfills a certain property but is not capable of deciding if the system (to be verified) is modeled appropriately for that purpose. In this thesis, *validation* is mainly used to assess if models of physical parts of the system and/or the environment are realistic.

2.3 Model-Checking

Model checking (or property checking) is a formal verification technique based on models of system behavior and properties, specified unambiguously in formal languages (see, e.g., [22]). The behavioral model of the system under verification is often specified using a Finite State Machine (FSM), in our case using synchronous FSMs. Their expressiveness is sufficient for the purpose of this thesis, but Petri nets, e.g., could be used as well, if needed.

The properties to be checked on the behavioral model are formulated in a specific property specification language. Several tools (such as SPIN [79], NuSMV [66] or nuXmv [27]) exist for performing these checks by systematically exploring the state-space of the system. When such a tool finds a property violation, it reports it in the form of a counterexample.

Although in this thesis, manly *Linear Time Logic* (LTL) is used for defining properties on models, first, the temporal Logic CTL* is introduced. CTL* "is a propositional modal logic with path quantifiers, which are interpreted over states, and temporal operators, which are interpreted over paths."[33, 1.2.2]. It is comprised of path quantifiers, temporal operators, and atomic propositions.

In total, there are two path quantifiers:

- $A\varphi$ ... φ hold for every infinite path from this state
- $E\varphi$ ... φ holds for some infinite path from this state

CTL* does define four temporal operators. However, since this thesis uses only two of them, only those are given here:

- $F\varphi$ ... φ holds at some state in the future
- $G\varphi$ ... φ holds in all states in the future

Using this one may comprise entire CTL* properties such as:

- *AG* (Always Globally): an expression p is true in the initial state s_0 and in each state of all transitions $s_0 \to s_1 \to s_2 \to \cdots \to s_n$.
- *EF* (Eventually in the Future): for an expression p and an initial state s_0, there exists a state sequence $s_0 \to s_1 \to s_2 \to \cdots \to s_n$ such that p is true in s_n.

The *Linear Time Logic* (LTL) "is the syntactic fragment of CTL* that contains no path quantifiers except a leading A" [33, 1.2.4]. Normally, LTL properties are given without the leading A. Thus, $AG\,p$ becomes $G\,p$. However, the CTL* property $EF\,p$ cannot be expressed as LTL formula.

For details on both CTL* and LTL this thesis refers to the Handbook of Model Checking [33] and its primary sources [17, 67]. However, to understand the remainder of this thesis, especially the properties used in this thesis, it is not necessary to have a deep understanding of those logics.

2.4 Counter-example Guided Abstraction Refinement

Counter-example Guided Abstraction Refinement (CEGAR) [32, 30, 31, 29] is an approach that uses a special form of abstraction called *over-approximation*, to reduce the state space in order to allow model checking of complex systems. Intuitively, an abstract model is an over-approximation of a concrete model, if it allows for all the behavior of the latter and possibly more. In the course of an abstraction, states of the concrete model are clustered into abstract states. This may already lead to an increase of behavioral options through the transitions between clustered states in the abstract model. However, no transition in the abstract model must be removed so that a possible behavioral option in the concrete model is not available in the abstract model. Over-approximation guarantees that, if a temporal logic expression in *ACTL** [3] evaluates to true in an abstract model, then it is true in the concrete model as well. If it evaluates to false in the abstract model, however, no conclusion can be drawn for the concrete model in this regard.

One drawback of over-approximation is that, if the abstract model violates an *ACTL** property we do not know whether the (concrete) system or the abstraction caused the violation. Whenever the model violates the property the model-checker generates a counterexample. Clarke et al. [31] uses it to first determine whether this counterexample is a valid path of the unabstracted model. If not, the information provided by the counterexample is used for refinement of the abstract model.

[3] *ACTL** is a subset of *CTL** allowing only the path quantifier A

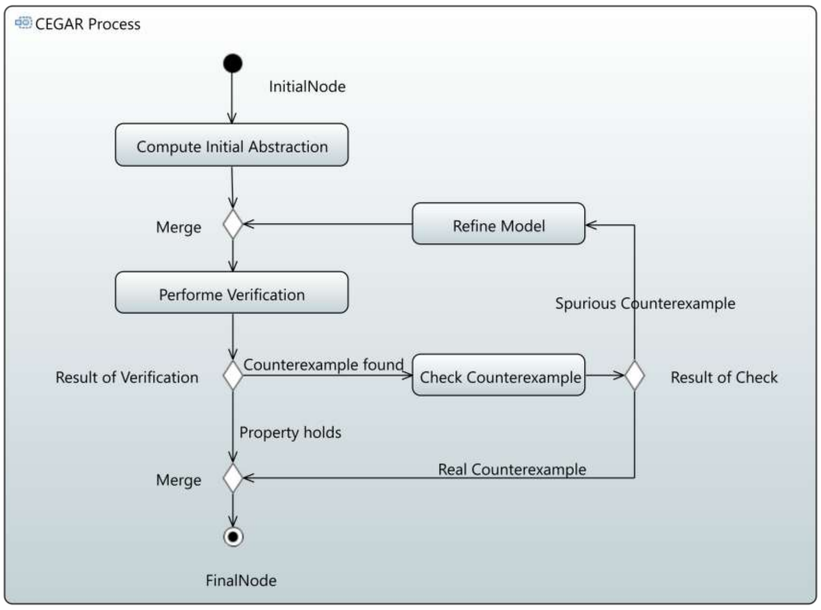


Figure 2.4: Workflow of CEGAR as defined in [31]

The activity diagram given in Figure 2.4 shows the overall workflow of CEGAR as defined by Clark et al [31]. The workflow starts with the computation of an initial abstraction. In this step, a coarse abstraction is generated. This is done by both taking the original model and the property to be verified into account.

In the *Perform Verification* step, the abstract model is checked against the given property, resulting either in a confirmation that the property holds or a counterexample. In the former case the workflow ends with *pass*, in the latter the workflow proceeds with the next step *Check Counterexample*.

Check Counterexample checks whether an abstract counterexample generated during the step *Perform Verification* is real or not (spurious). A abstract counterexample is said to be real if it corresponds to an actual counterexample in the unabstracted model. The check is done by simulating the counterexample on the unabstracted model and has two possible outcomes. If the counterexample is real (the abstract counterexample is realizable in the unabstracted model) the CEGAR workflow later ends with *fail*. If the counterexample is spurious, the workflow is proceeding.

The step *Refine Model* uses information from the spurious abstract counterexample and the preceding step, to refine the abstract model in such a way that the given counterexample is no longer admitted by the refined model. This model is then used in the following *Perform Verification* step.

Actually, Clark et al. [31] define CEGAR consisting only of three steps – *Generate the initial abstraction*, *Model-check the abstract structure* and *Refine the abstraction*. However, for our representation of the workflow, we decided to make the step of checking whether a counterexample is real or spurious (*Check Counterexample*) explicit.

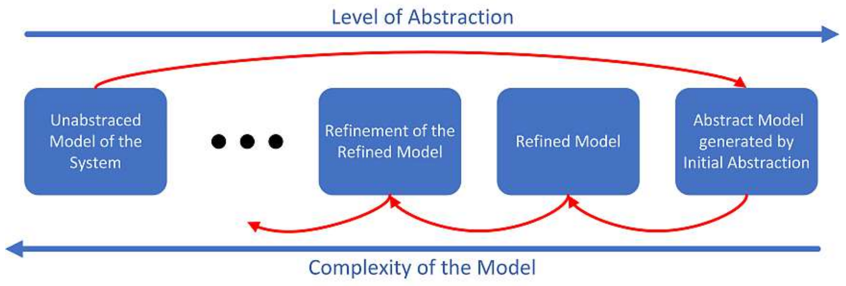

Level of Abstraction
Unabstraced Model of the System
• • •
Refinement of the Refined Model
Refined Model
Abstract Model generated by Initial Abstraction
Complexity of the Model

Figure 2.5: Model hierarchy; the red arrows indicate the order in which the models are computed.

Figure 2.5 shows the models generated during the execution of the workflow. Each arrow represent an entire cycle of verification, checking the counterexample and refinement of the model. The model used in the verification step gets less abstract from cycle to cycle and, therefore, harder to verify. Theoretically, the cycle could be executed until the refined model ends up to be the unabstracted model again. However, this is unlikely since the process is likely to prove or disprove the property before.

In essence, CEGAR is a workflow to verify *ACTL** properties by first generating a coarse abstraction of the model and gradually refining the abstraction until the given property holds in the abstract model (and, therefore, the unabstracted model) or it finds a counterexample that exists in the abstract model as well as in the unabstracted model.

Over the years, much work was done to adopt this idea for different types of models, like graph transformation systems [54] and/or for using it in various domains, like classical planning [77, 78]. In this thesis, we also rely on this basic idea of using counterexamples for refining models.

CHAPTER 3

Qualitative Models and their Refinement

Model-driven software engineering often starts with abstract models of the intended behavior of the system to be built. To check whether the systems have certain properties and/or behavior, static and dynamic analysis, like model validation and executable models, may be used early on. However, since especially cyber-physical systems interact with external entities, not only the system to be built is to be modeled in order to perform those checks. This raises the challenge of modeling those entities on the right level of abstraction.

In this chapter, we show means of modeling cyber-physical systems as abstract models, qualitative models, and refining them in this case to more concrete models, which are quantitative models. The difference between those two types of models lies in the use of concrete values. Qualitative models do not use quantitative values (e.g., speed values).

The remainder of this chapter is organized in the following manner. Since our running example comes from a research project regarding feature interaction, the chapter starts with a short explanation of feature interaction and feature coordination. This section also motivates why we chose model-checking as the means of verification. After that, our running example, used in this chapter, adaptive cruise control (ACC), is introduced and formulated as a feature coordination problem. The following section shows the systematic workflow that we propose for mapping qualitative models to quantitative ones. Then we show our modeling approach for qualitative models and also give the models corresponding to the running example. In the subsequent section, we use this workflow to map those qualitative models to quantitative ones. The chapter concludes with related work and a short discussion of the results.

3.1 Background on Feature Interaction

In the beginning, research in the field of *features* and *feature interaction* was mainly done in the domain of telecommunication systems. In this domain, a feature was defined as a "package of incrementally added functionality providing services to subscribers or the telephone administration"[26]. Later, the term feature was defined in a broader sense by the IEEE Standard for Software and System Test Documentation [1], as "A distinguishing characteristic of a system item (includes both functional and non-functional attributes such as performance and reusability)". This definition was then brought into further areas through the adoption by the IEEE standard *Systems and software engineering - Vocabulary* [2].

Both definitions indicate that a system can be composed of (more or less independent) entities that group functional and non-functional characteristics. Those entities are called features.

However, in general, features are not independent of each other. In a system, features can interact with each other in many ways. Some interactions are intended, for example, to achieve a complex behavior. And some interactions are inadvertent because they cause undesired behavior of the system or critical system states [18].

None of the mentioned standards define the term/phrase *feature interaction*. However, Bowen et al. [26] define feature interaction with the following statement "Feature interaction occurs when one feature affects the operation of another feature (whether in terms of system specification, design, implementation, or execution)". Later, Apel et al. [18] defined it as the emergent behavior that cannot be deduced from the behavior of the individual features involved in the interaction. The difference between their two definitions lies in the scope. Whereas Apel et al. [18] only look at the behavioral aspect of the system and features, Bowen also considers other aspects of the software. For the remainder of this thesis, we use the more narrow definition of Apel et al. [18].

For an intuitive view of features and feature interaction, let us consider an example from the telecommunication domain. Let us suppose we have a very simple mobile phone. This phone has two features, the handling of voice calls and Short Message Service (SMS) text messages. Each feature fulfills a different user need, the need to be able to make telephone calls and the need to be able to send and receive text messages.

At the first glance, those features look independent from each other, but at a closer look that its not entirely true. Let us suppose that the user of the mobile phone has currently a phone call. At the same time, there is an SMS message arriving. Normally, this would trigger a loud notification sound generated by the speaker dedicated for ring-tones. However, since the user has currently a phone call, the loud sound would at least disturb the ongoing call. Because a regular user would not like such behavior, this is a problem. To solve it in our simple phone, an intended feature interaction was introduced by design. This interaction causes that instead of a loud notification sound a softer single tone informs about the incoming message.

This example shows that in most cases the behavior that emerges by simply adding features to an already existing system (consisting of features) is not the most desired one.

We also chose this example to show that feature interaction can not only occur in the system itself but in its environment as well. This type of feature interaction is characterized by not causing any internal conflict.

Verification of *feature interactions* (FIs) and their coordination is important [18] especially if features are used in domains that are considered safety-critical, like the domain of automotive vehicles.

Since more and more features are integrated in such systems, three major challenges emerge:

- Detecting possibly undesired feature interaction,
- resolving feature interaction by means of feature coordination,
- and proving that features combined with their coordination behave as desired.

In particular, verification of feature coordination is important with regard to various technical standards like the functional safety standard for road vehicles ISO 26262 [50]. This ISO standard formulates a high standard on how the behavior of automotive systems has to be verified. In contrast to testing, model-checking has at least in theory the advantage to prove the correctness of a given criterion. Therefore, we chose model-checking for verification.

This chapter of the thesis focuses on coordinating features with known interactions. Even if these are known, coordinating the features involved poses additional challenges [51].

3.2 Running Example – Adaptive Cruise Control

Adaptive Cruise Control (ACC) is a feature of premium class vehicles and is also frequently installed in medium-class vehicles and is considered the successor of the *Cruise Control* (CC) feature. CC merely maintains a predetermined speed until the feature is turned off explicitly or by applying the brakes. Therefore, to adapt to shifting traffic conditions, the driver must manually change the preset target speed or deactivate CC entirely. ACC enhanced the functionality of CC to handle a wide variety of traffic situations by automatically changing the vehicle speed. Mostly this is done by using radar sensors to determine the distance to other vehicles in front and their speeds.

Figure 3.1 gives a scenario showing different operation modes of ACC. First, the blue vehicle is on a completely clear road. In this situation, ACC works the same as CC and keeps the vehicle at a preset speed. Further down the road (scenario 2 in Figure 3.1), some traffic in the form of the red vehicle is around. Since this vehicle drives more slowly than, the set speed of the blue vehicle, some intervention is needed to avoid a collision. Therefore, ACC switches in its mode for *Distance Control* (DC). This mode constantly adopts the vehicle's speed to the speed of the (red) vehicle in front. If it slows down, the ACC-controlled (blue) vehicle slows down automatically. When the vehicle in front accelerates and, therefore, moves away, the ACC system accelerates up to the same speed

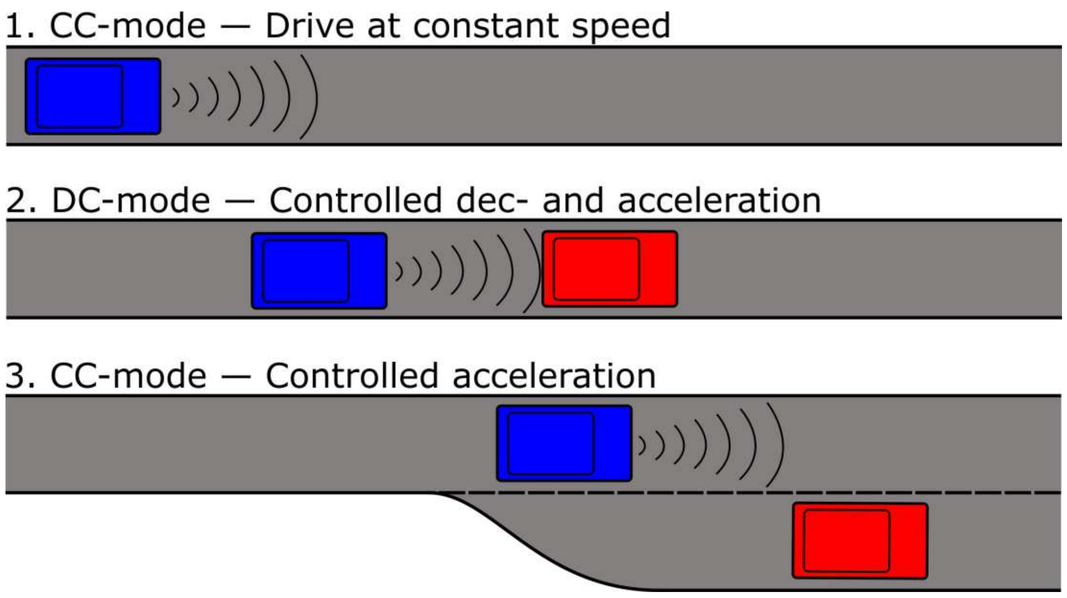

1. CC-mode — Drive at constant speed
2. DC-mode — Controlled dec- and acceleration
3. CC-mode — Controlled acceleration

Figure 3.1: Illustration of a scenarios during which ACC is in different operation modes (drawn after [13])

(unless the speed is greater than the preset target speed of CC). Basically, in DC-mode, ACC controls the speed in such a way that it always matches the speed of the (red) vehicle in front. After the red vehicle leaves the shared lane (scenario 3 in Figure 3.1), the blue car switches back to CC-mode since there is no vehicle in front. However, since the current speed is lower than the preset speed, ACC performs a controlled acceleration till the vehicle reaches the latter.

For the remainder of the chapter, we study ACC from the perspective of a *composite feature* that includes both features, *Cruise Control* (CC) and *Distance Control* (DC). In addition, this thesis considers only speed requests are issued by the features. Of course, this is a simplification, since in reality a certain acceleration would have to be requested that is intended to lead to a requested target speed.

However, simply running CC and DC alongside each other does not produce the described behavior. Since both features each request a certain vehicle speed and different factors influence these requests, a feature interaction is inherent. To resolve this feature interaction and actually achieve the desired behavior, we need some sort of coordination. For our running example, we chose to use a dedicated coordinator to do so.

Figure 3.2 shows how the *composite feature* ACC is composed from the features CC and DC as well as a dedicated *Coordinator* (CO). The figure also shows the flow of data between the components and beyond the borders of ACC. Whereas CC only uses the set target speed, DC uses a set target distance, the distance measured by a sensor, and an estimated value for the speed of the red vehicle as the basis for calculating its speed request. The *Coordinator* then calculates the final speed request based on those two speed requests.

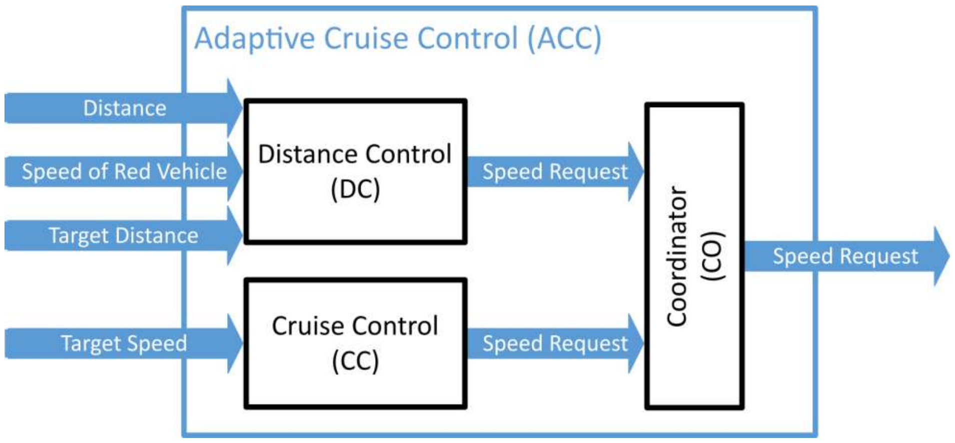


Figure 3.2: Block Diagram of Adaptive Cruise Control as a combination of the features Cruise Control and Distance Control

3.3 Systematic Top-down Design with Integrated Validation and Formal Verification

Now let us explain the proposed workflow for systematic top-down model design we propose to map qualitative models to quantitative ones. For a better overview of the whole workflow, a graphical representation is given in Figure 3.3. It gives the overall structure of the workflow and also shows the order – possibly depending on the outcome of the execution of an activity – in which the activities are executed.

The individual activities in this workflow are divided into three categories – activities performed on the qualitative, quantitative, and both models. The left and right blue zone in Figure 3.3 give all activities performed on the qualitative and quantitative models, respectively. E.g., model-checking is only done on the qualitative model. The activities we deem cross-cutting in the sense that both models are involved are given in between those (blue) zones. The most important is determining if the qualitative model is an over-approximation of the quantitative model.

The workflow starts with creating an initial qualitative model (if one does not already exist). This activity requires design skills and domain knowledge as well as a vision. Still, it should be easier for a domain expert to create such a qualitative model. We will later see what exactly is meant by a quantitative model.

Model-checking the qualitative model may already, at this stage, reveal property violations in the form of counterexamples. Since, at this point in time, technically, there is no mapping and, therefore, no quantitative model, it is pretended that all counterexamples are real ones. However, each counterexample has to be analyzed in order to gain information on why a particular property was violated. The information gained is then used to modify the qualitative model manually. The new model is the new input to the model-checker. This cycle of model-checking, analysis of the counterexample, and modification is done until model-checking returns no counterexample.

For a verified qualitative model, a mapping to a quantitative model is to be determined.

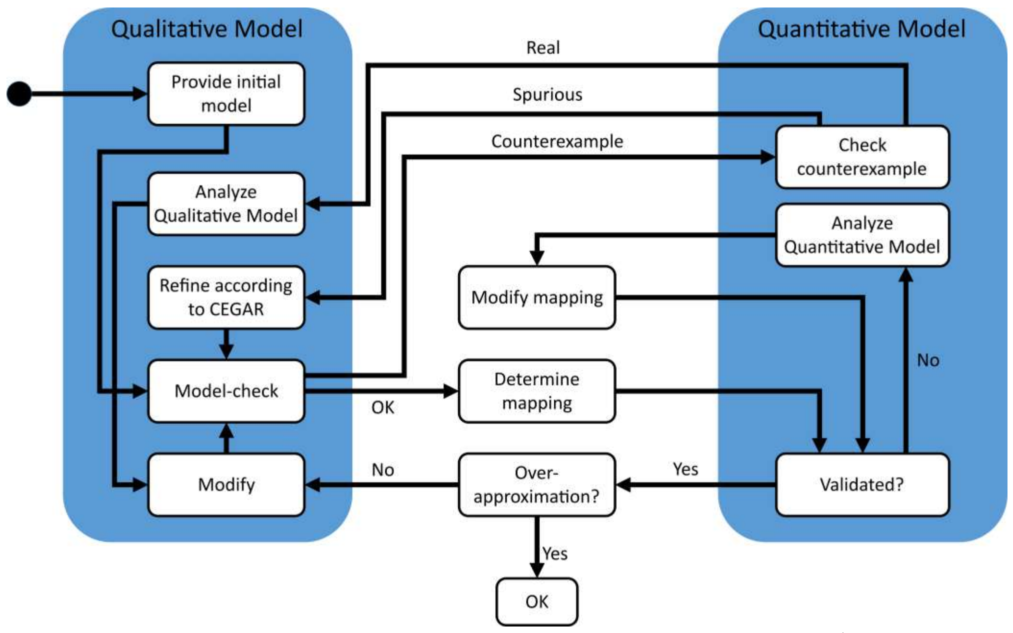


Figure 3.3: Proposed workflow for systematic top-down model design (revised version of [59, Fig. 1])

This step also requires domain knowledge. However, this step may not be entirely manual since, in certain domains, it might be possible to encode the required knowledge in dedicated software.

The resulting quantitative model should be validated eminently. The main focus of this validation is to assess if the mapping and the resulting model are realistic in their assumptions about the physical environment the CPS is embedded.

Since a failed validation shows some sort of misconception about the system and/or its environment a detailed analysis of the quantitative model has to be done. The analysis has the goal of showing where in the qualitative model the misconception has an impact. With this information, the mapping can be altered accordingly. However, since it is not guaranteed that the quantitative model is valid, the cycle of validation, analyzing, and modifying is redone until the model is valid.

The quantitative model must be a *over-approximation* of the qualitative model in order to determine whether the quantitative model satisfies the properties that the qualitative model has already been successfully verified against. This needs to be proven. Performing such a proof manually can be challenging but has to be done in any case. The more obvious case is when the mapping was altered after a failed validation. However, even if the validation immediately after the mapping was determined succeeds, the resulting quantitative model may not be an over-approximation due to mistakes made during defining the mapping.

If the bottom-up over-approximation check has been successful, the quantitative model is verified successfully against the same properties that hold for the qualitative model. Hence the workflow can stop. However, if the over-approximation check fails, the qualitative model needs to be modified accordingly and model-checked again.

If model-checking reveals a counterexample, it is checked whether it is *spurious* (according to [31]). This is the case if it is not a counterexample for the quantitative model, i.e., it is an artifact of the abstraction. Hence, the abstract model is to be *refined*, and this can be done systematically according to [31, 4.4]. The qualitative model must also be altered if the counterexample is considered real. However, a detailed analysis of the counterexample may be needed to spot the cause of the issue and fix it. This cycle is done until the qualitative model is verified successfully again.

The overall workflow is executed until a successfully verified qualitative model is mapped to a quantitative model that is considered valid, and the check for over-approximation is passed.

3.4 Modeling Approach

To better understand the workflow, this section first gives a more elaborate explanations of what qualitative models, quantitative models, and mappings are. Later an example is used to explain how to determine if a qualitative model is an over-approximation of a quantitative model.

Qualitative Model In the context of our workflow, a qualitative model is a UML State Machine consisting of states and transitions between those states. The states have some meaning attached to them. In this sense, we consider them as qualitative states. As usual, the transitions represent possible changes in the current state of the model. It is important to note that the qualitative model does not contain or in any form reference concrete values and/or mathematical formulas.

Mapping The mapping defines to which point or areas of the state space a qualitative state is mapped. In its simplest form, it specifies that a qualitative state represents a specific variable holding a specific value, e.g., $distance = 25\,m$. However, a qualitative state may also be mapped to a whole value range, e.g., $20\,m \leq distance < 30\,m$.

Quantitative Model The quantitative model is a UML State Machine with the same number of states. However, those states represent a specific point or area in the (more concrete) state space of the system. The corresponding systems determine the transitions between those states.

For parts modeling the CPS, we assume that the (subsequent) implementation conforms to the qualitative model. Hence it allows only state transitions already included in the qualitative model. Therefore, the transitions taking into account the mapping. For parts modeling the environment of the CPS, we cannot assume that it conforms to the qualitative model. The behavior of the environment determines the transitions. Based on the mapping, an analysis has to be done on which transitions are possible and which are not.

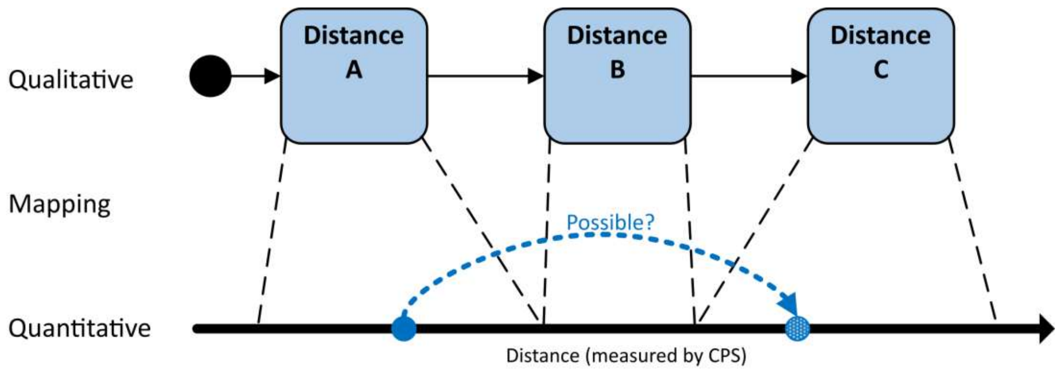


Figure 3.4: Mapping of qualitative states to the domain of real value

To illustrate the relationship between those concepts, Figure 3.4 uses the simple example of the physical property distance. The qualitative model of our example models the assumption a CPS has over this property and only defines three distance states – Distance A, Distance B and Distance C. The transitions give that the distance may only increase step by step. Thus, a changeover from Distance A to Distance C has to go over Distance B.

The states of the qualitative model are mapped to the "real world" by mapping each state to a whole range of (quantitative) values – with meter as the unit of measurement – as illustrated by the black broken lines.

Over-approximation is only guaranteed when all possible changes in the "real world" are covered by at least one transition of the qualitative model. Assuming that the distance can only increase, this is the case at first glance since the physical property distance is a continuous-time signal. However, due to sampling, it may appear to the CPS that the distance "jumps" from a distance classified as Distance A to a distance classified as Distance C – given as the blue dotted line. Suppose such a "jump" is possible in our example. In that case, the qualitative model is not an over-approximation of the quantitative model since there is no direct transition from Distance A to Distance C. Therefore, checking whether the qualitative model is an over-approximation comes down to determining all possible changes the CPS may perceive and checking if the qualitative model has a corresponding transition for all of them.

3.5 Qualitative Model of ACC

To create a comprehensive initial qualitative model of ACC, we deem it necessary to first gain some insight into the physics of the system. We did this by visualizing the system's physical chain. Figure 3.5 gives this visualization, there physical quantities are given via elliptic shapes. The relationship between speeds – Speed A and Speed B – and distance is given via the integrator. It also shows that some initial value, in our case the Initial Distance is needed. Since we are particularly interested in verifying the *composite feature* ACC, it also includes the feature model. The broken arrow in the figure indicates that

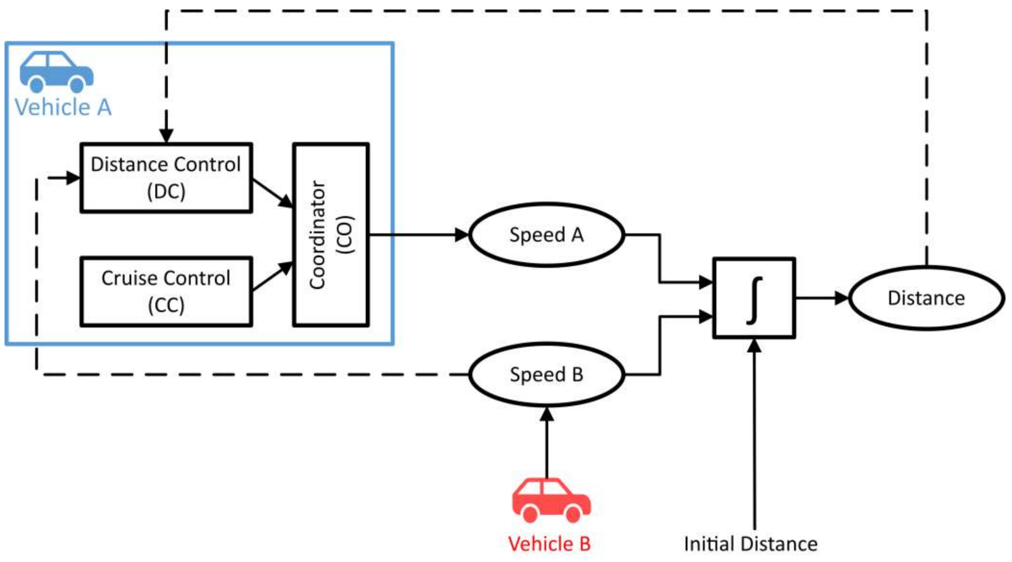


Figure 3.5: Physical chain together with feature model

measured distance value (in reality provided by a sensor) and an estimation of Speed B are fed back to DC, so that DC can keep control on a target distance. More precisely, a closed-loop controller implements DC [70].

Our approach was to model all of the physical quantities – Speed A, Speed B, and Distance – as well as all the behavior of the essential software components – Distance Control, Cruise Control, and Coordinator – as UML State Machines. However, we realized that the Coordinator directly influences Speed A. Therefore, Speed A was not modeled as a FSM on its own. Instead, the speed requested by the Coordinator is taken as the value for Speed A. Therefore, the model consists only of five FSMs, namely:

- Distance Control (DC) model,
- Cruise Control (CC) model,
- Coordinator (CO) model,
- Vehicle B model, and
- Distance Classification model.

Figure 3.6 provides an overview of those FSMs. However, for presentation reasons, this figure only gives details for the two FSMs representing the physical quantities Speed B (Vehicle B) and Distance (Distance Classification). (Details on the remaining three FSMs can be found below in Figures 3.7 – 3.9, but those are not important for the moment.)

Since our approach is a top-down approach, those FSMs do not use concrete values for physical quantities but qualitative "values" instead. For our initial models, three qualitative "values" –*Low Speed*, *Medium Speed*, and *High Speed* – are used to model

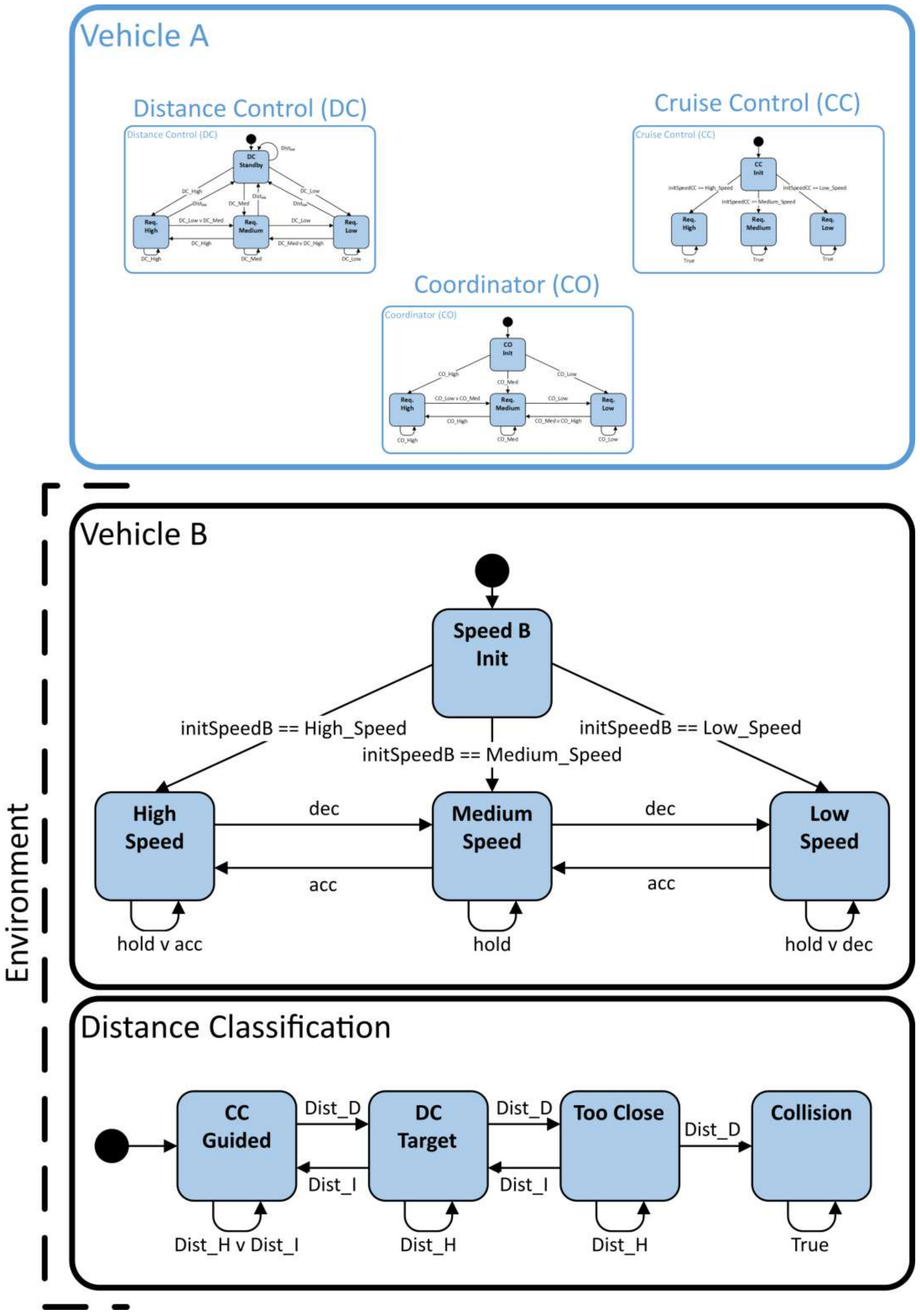


Figure 3.6: State machine modeling ACC and its physical environment – Some text is not readable by intention.

speeds. For modeling distances, the “values” *CC Guided*, *DC Target*, *Too Close* and *Collision* are used.

For a better understanding of the model, in the following, we provide more details of the individual FSMs.

Vehicle B The state machine of Vehicle B given in Figure 3.6 models the behavior of the vehicle in front and, therefore, the physical quantity Speed B. It is assumed that Vehicle B is out of control of Vehicle A. We also assume that Vehicle B cannot abruptly change its speed from *High Speed* to *Low Speed* nor in the other direction. However, it can randomly accelerate (acc), decelerate (dec), or hold its speed. The initial speed is also randomly chosen. For model-checking, this is encoded via variables representing the action the vehicle takes in each step and the initial speed, respectively [70].

Distance Classification The FSM Distance Classification models the current distance between Vehicle A and Vehicle B. Instead of actual distance values the qualitative “values” *CC Guided*, *DC Target*, and *Collision* are used. *CC Guided* represents a range of concrete values where the vehicle upfront is considered far away, compared to the target distance and therefore Distance Control (DC) is deactivated. The state *DC Target* represents all system states where DC is active and adapts the speed of Vehicle A to the speed of Vehicle B. The state *Collision* represents a collision between Vehicle A and Vehicle B. The transition conditions are given in Table 3.1. For example, if the speed value of both vehicles is currently low (i.e, *Low Speed*) then the distance is kept. This corresponds to $(LowSpeedA \wedge LowSpeedB)$ and, therefore, the expression *Dist_H* becomes true. All other expressions evaluate to false.

Table 3.1: Conditions for Transitions of FSM of the Distance Classification according to [70]

Name	Condition
Dist_H	$(LowSpeedA \wedge LowSpeedB) \vee (MediumSpeedA \wedge MediumSpeedB)$ $\vee (HighSpeedA \wedge HighSpeedB)$
Dist_D	$(MediumSpeedA \wedge LowSpeedB) \vee (HighSpeedA \wedge LowSpeedB)$ $\vee (HighSpeedA \wedge MediumSpeedB)$
Dist_I	$(LowSpeedA \wedge MediumSpeedB) \vee (LowSpeedA \wedge HighSpeedB)$ $\vee (MediumSpeedA \wedge HighSpeedB)$

Below, we will use compressed representations like the one in Table 3.2. In this representation, each row contains a different speed value of Vehicle A, and each column a different speed value of Vehicle B. The corresponding table entry specifies which condition(s) evaluate as true. Generally, the entry in each cell of Table 3.2 corresponds to the row of Table 3.1 where the term $(xSpeedA \wedge ySpeedB)$ appears [60].

Table 3.2: Transition conditions of the Distance Classification FSM – compressed version of Table 3.1 (taken form [60])

Speed A\B	Low	Medium	High
Low	H	I	I
Medium	D	H	I
High	D	D	H

H . . . Hold distance (Dist_H)
I . . . Increase distance (Dist_I)
D . . . Decrease distance (Dist_D)

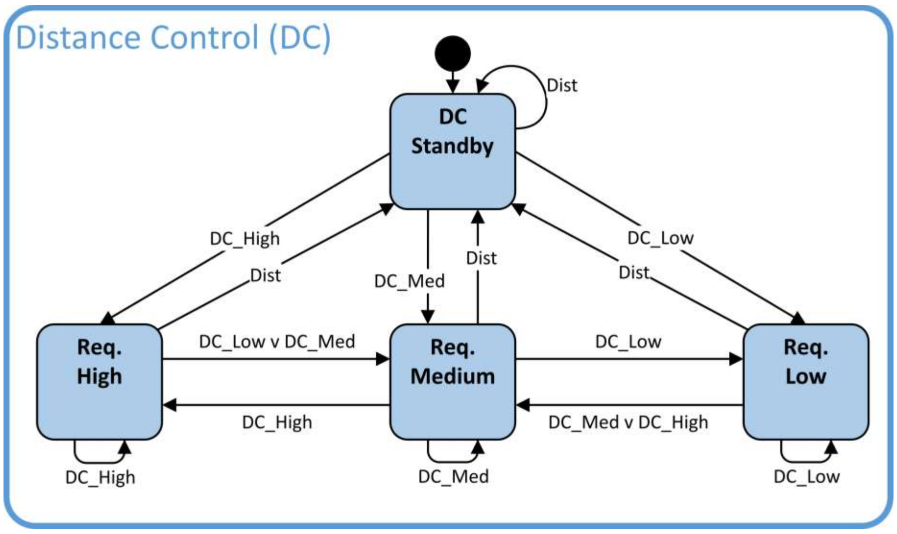


Figure 3.7: FSM of Feature DC (adopted version of [70, Fig. 2]

Distance Control (DC) [70] Figure 3.7 depicts the FSM that implements a simple closed-loop controller within the context of our qualitative model, with Table 3.3 defining the Boolean expressions used in the diagram. The target is the (constant) target distance specified for DC. This FSM accepts as input values that model the distance measured by a sensor and the estimated speed of Vehicle B. Since measuring the distance and estimating the speed takes time, our model accounts for that by introducing a time delay. In the nuXmv [27] implementation, this is modeled via one-step delay between the Distance Classification and Speed B having particular values and DC seeing those values. To indicate those delays in the condition table the suffix “del” is used.

This FSM for modeling DC has four states. The initial state is DC Standby. This state is entered directly after initialization. However, it is kept or transferred to whenever the distance is greater than the target distance (in the FSM this is modeled via *Dist* is true). In this state, DC does not issue any speed requests. In the other three states, DC requests one of the possible speed values. E.g., *Medium Speed* is requested whenever the FSM is in state *Req. Medium*.

Table 3.3: Transition conditions related to the DC FSM (extended version of [60, Table 3])

Distance$_{del}$ \Speed B$_{del}$	Low	Medium	High
CC Guided	Dist	Dist	Dist
DC Target	DC_Low	DC_Med	DC_High
Too Close	DC_Low	DC_Low	DC_Med

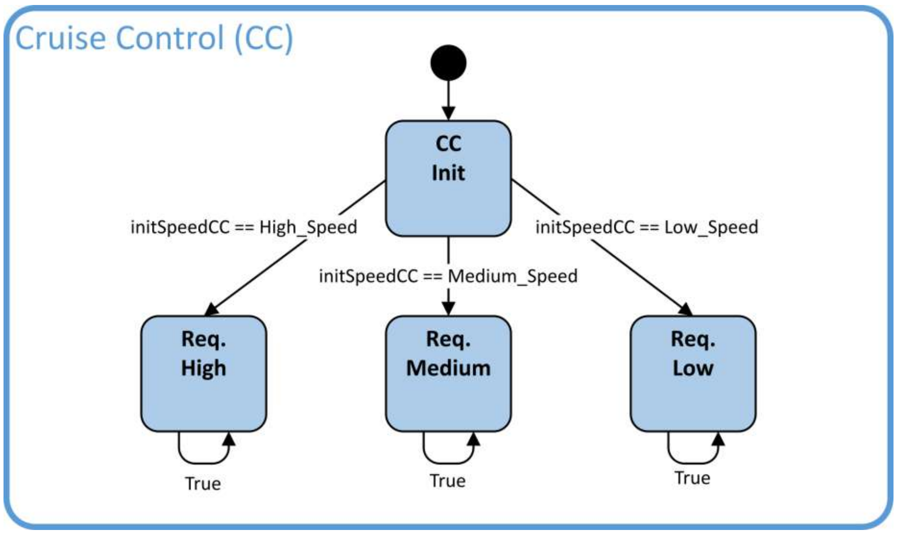


Figure 3.8: FSM of Feature CC (adopted version of [70, Fig. 3]

The condition *DC_Low* checks whether a request for *Low Speed* would be in order. Analogously *DC_Med* and *DC_High* check for Medium and *High Speed*, respectively. However, according to our modeling approach, speeds can only change to directly adjacent values. Therefore, it is not always possible to issue the corresponding request directly. The FSM models this by making it impossible to directly go from *Req. Low* to *Req. High* and vice versa. Such changes have to head through *Req. Medium*.

Cruise Control (CC) [70] Compared to DC, the behavior of CC is relatively simple. As shown in Figure 3.8, the FSM also has four states. However, the possible transitions between those states are limited. After CC is initialized (state *CC Init*), the given target speed determines the transition to one of its three remaining states. After that, it simply remains in this state and requests the same speed. We assume that the target speed (initSpeedCC) is set by the vehicle's driver and, therefore, out of the system's control.

Coordinator (CO) [70] For our investigations, the coordination of the two features, CC and DC, needs to be modeled. Here, we include a coordinator according to the approach described in [84]. In essence, the approach described in [84] takes the minimum of the acceleration values that cruise control and distance control request, and ACC

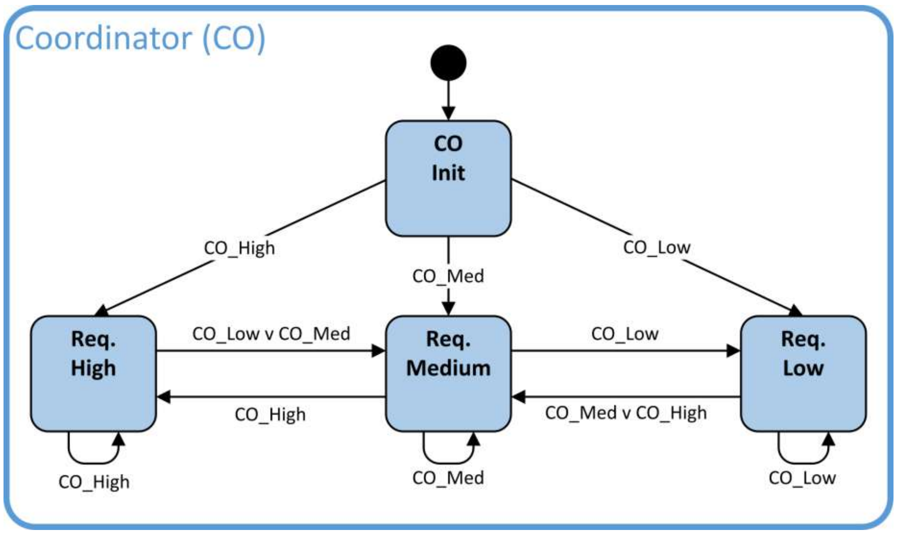


Figure 3.9: FSM of the Coordinator (CO) (adopted version of [70, Fig. 4])

requests this minimum acceleration. Since the features in our approach request speed values, our coordinator takes the quasi-minimum of speed requests instead. We call that a quasi-minimum because DC does not request any speed when it is in DC Standby. Therefore, the coordinator has to handle this as well. In Table 3.4 the corresponding cases are represented in the first row labeled with *No Req.*.

After initialization of the coordinator (for the FSM see Figure 3.9), the first speed request is issued based solely on the conditions. However, after that, the quasi-minimum of those requests depends also on the current state of the coordinator. The reason is the same as for DC. Our modeling approach does not allow direct transitions from *High Speed* to *Low Speed* and vice versa.

Table 3.4: Transition conditions related to the coordinator FSM given in Figure 3.9 (extended version of [60, Table 4])

Req. DC\CC	Low	Medium	High
No Req.	CO_Low	CO_Med	CO_High
Low	CO_Low	CO_Low	CO_Low
Medium	CO_Low	CO_Med	CO_Med
High	CO_Low	CO_Med	CO_High

Property Since we now have the whole qualitative model, we can define properties on it. In general, properties may be formalized in many different ways, depending on the used model-checker and the algorithms used for verification. nuXmv provides many different logics to formalize those properties. However, we focus on CTL and LTL only.

Since we are particularly interested in verifying if ACC is safe in the sense that it avoids rear-end collision accidents with vehicles in front, we defined the following formula for verification [70].

$$\varphi_{CTL} = AG(state\_phy\_dist \neq COLLISION) \tag{3.1}$$

Basically, it says that it is always globally true that the physical distance (state of the Distance Classification FSM) is different from a collision.

In contrast to UML State Machines, nuXmv allows choosing the initial state based on conditions. For example, it is possible to encode cruise control in such a way that it is immediately in one of the states requesting a speed – *Req. High*, *Req. Medium* and *Req. Low*. Therefore, instead of waiting for the initialization to finish, a request can be already sent in the first step of verification. We have used this possibility to allow us to simplify the synchronization between the FSMs. Keep this in mind for the rest of the chapter, especially when counterexamples are listed and described.

Now that we have our initial qualitative model of the whole system, we can start with our case study in the next section.

3.6 Case Study and Results

Our case study started with the qualitative ACC model presented in the last section. Therefore, there will be references to that section, especially to the figures and tables shown. This qualitative model has been successfully verified through model-checking against the rear-end collision accident property 3.1 .

Actually, our first draft of the qualitative model (see [70]) did not include the state *Too Close* in the Distance Classification FSM. Model checking this first draft led to a violation. Based on the generated counterexample, we learned that an additional state in the distance classification model is necessary for a realistic qualitative model, which we denoted as Too Close [70]. Hence, we deem it essential that the first mapping (from qualitative states to quantitative values) is done only after the (initial) qualitative model is verified successfully. To guarantee this, the workflow calls for model checking the initial model.

In the following, all tables regarding transition conditions belong to the qualitative model. Hence, all changes in these tables or the FSMs are changes in the abstract model. To better recognize the changes between a table and its replacement, differences are marked in orange. In contrast to tables associated with the initial model, tables in this section may contain more than one entry per cell. If this is the case, all of the mentioned transition conditions evaluate to true simultaneously. Hence, this introduces non-determinism.

3.6.1 First Mapping to a Quantitative Model

Our first mapping was from the qualitative ACC initial model, which we explained in Section 3.5, to a quantitative model. According to our mapping approach outlined above, we first selected the total range of speed values and partitioned it according to the number of speed classes in the qualitative model. More precisely, we defined three concrete speed values ($15\,m/s = 54\,km/h$, $22.5\,m/s = 81\,km/h$, and $30\,m/s = 108\,km/h$), corresponding to the three speed values defined in the initial model (*Low Speed*, *Medium Speed*, and *High Speed*). Since we selected the same concrete speed values for both vehicles, there is no change in their distance in the quantitative model whenever they are in the same speed class.

Then we defined the cycle time for the model-checking runs as $1s$. According to the qualitative model, a change from one speed class to an adjacent class is allowed in one time-step. Consequently, the possible speed changes are $-7.5\,m/s$, $0\,m/s$, and $+7.5\,m/s$, i.e., possible accelerations of $-7.5\,m/s^2$, $0\,m/s^2$, and $+7.5\,m/s^2$.

Based on that, we mapped the qualitative model's Distance Classification to real-world distances in our first quantitative model. In particular, the maximum speed difference between the two vehicles of $15\,m/s$, results in a maximum change in their distance in a single time-step of $15\,m$. Taking the constraints mentioned above on such a mapping into account, we defined the following mapping of distances:

- CC Guided = $(35\,m\ ,\ \infty]$
- DC Target = $(20\,\mathrm{m}\ ,\ 35\,m]$
- Too Close = $(0\,m,\ 20\,m]$
- Collision = $[-\infty,\ 0\,m]$

Furthermore, the Newtonian equations for the physics involved, such as

$$Distance_{n+1} = (SpeedB - SpeedA) * \Delta t + Distance_n \tag{3.2}$$

had to be provided to the quantitative models. They serve as the foundation for validation during the investigated workflow.

One may ask why the quantitative model does not include some numerical equation for the DC controller. The main difference between the DC controller and physics (Distance Classification) is that the controller can be designed top-down. Therefore, we assume that the concrete numerical equation can be specified after the workflow is finished. However, the equation must comply with the assumptions made during the workflow. In contrast, physical relationships cannot be changed. Therefore, the models must include them from the beginning to ensure that they are not contradicted.

3.6.2 Detailed Analysis and Model Modification

Analysis of the physical behavior of the quantitative model resulting from our first mapping showed that the original qualitative model does not fully cover its behavior. For example, when Vehicle A drives *Low Speed*, Vehicle B drives *Medium Speed*, and their distance is classified as *DC Target*. According to physics, it is not necessarily in the case that the distance changes to a value that is classified as *CC Guided*. Depending on where it was previously, the distance can also be kept within an interval of *DC Target*. To fill the example with concrete values, we choose $22\,m$ as the distance before the step is taken. From the mapping of speeds, we know that *Low Speed* and *Medium Speed* correspond to $15\,m/s$ and $22,5\,m/s$, respectively. Hence, a difference of $7,4\,m/s$ and therefore a change of the distance of $7,5\,m$ per step. Altogether, this results in a distance of $29,5\,m$, which is also within the range of *DC Target*.

Based on our analyses, the conditions must be as given in Table 3.5, where the letters in orange indicate the additional possibilities as compared to Table 3.2.

Table 3.5: Transition conditions according to physics with distance ranges ([60, Table 5])

Speed A\B	Low	Medium	High
Low	H	H/I	I
Medium	H/D	H	H/I
High	D	H/D	H

H . . . Hold distance (Dist_H)
I . . . Increase distance (Dist_I)
D . . . Decrease distance (Dist_D)

This adapted table indicates non-deterministic state transitions in the Distance Classification FSM at the bottom of Figure 3.1. This leads to a collision when model-checking the adapted qualitative model with the new transition conditions, with the following counterexample output by the tool NuSMV [66]:

1. Vehicle A and Vehicle B drive *High Speed* and *Medium Speed*, respectively. → Distance changes from *CC Guided* to *DC Target*.

2. Vehicle A drives *Medium Speed* (because of the DC's request) and Vehicle B accelerates to *High Speed*. → Because of the new transition conditions, the distance *DC Target* may be kept or increased to *CC Guided*. In this example, the distance is kept at *DC Target* (while in the previous model, the distance would change to *CC Guided*).

3. Vehicle A drives *High Speed* (DC recognizes *High Speed* of Vehicle B and the distance *DC Target* from the previous step) and Vehicle B decelerates to *Medium Speed*. → Distance changes to *Too Close*.

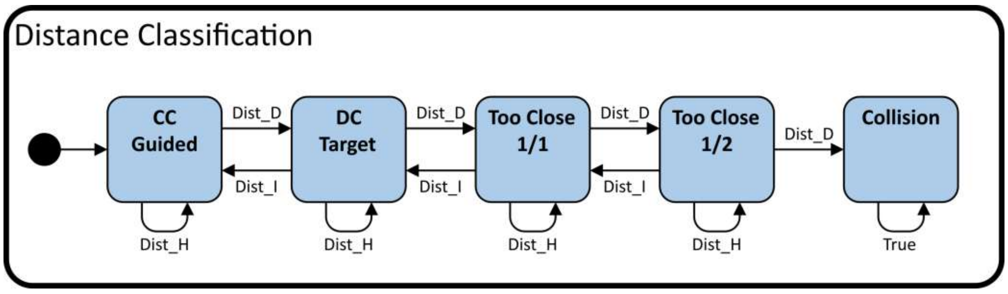


Figure 3.10: Distance Classification with *Too Close 1* split according to CEGAR refinement (adopted version of [60, Fig. 3])

4. Vehicle A decelerates to *Medium Speed* (the transition conditions even suggest *Low Speed*; however, a transition from *High Speed* to *Low Speed* is not possible) and Vehicle B decelerates to *Low Speed*. → Distance changes to *Collision*.

According to the workflow, the counterexample was checked to determine whether it is spurious or real. The analysis started from the end and has worked its way back.

In the last step (step 4) of the counterexample, it is revealed that before the collision ($Distance <= 0\,m$) occurs Vehicle A and Vehicle B drive with *Medium Speed* and *Low Speed*, respectively. Using the defined quantitative values together with Equation 3.2 we come to the following inequation $Distance_{n+1} = (15\,m/s - 22.5\,m/s) * 1\,s + Distance_n <= 0\,m$. Reshaping the inequation reveals that the ($Distance_n$) has to be in the range of $[0\,m,\, 7.5\,m]$ so that a collision is possible.

Analyzing step 3, we conclude (also based on Equation 3.2) that the distance before taking step 3 has to be in the range of $[7.5\,m,\, 15\,m]$ so that there is a possibility of a collision in step 4. However, this range does not overlap with the range associated with *DC Target*. This contradicts the counterexample. Thus, we conclude that the counterexample is spurious.

Since the check revealed that it is spurious, CEGAR refinement was applied, which led to the distance classification depicted in Figure 3.10. Changing the Distance Classification also triggered a change in the model of the DC controller. In particular, the transition conditions must consider the new state. This led to the conditions given in Table 3.6.

Model-checking the changed model did not reveal any collision [60].

Table 3.6: Transition conditions of the DC FSM corresponding to the Distance Classification with two *Too Close* states (extended version of [60, Table 6])

$\text{Distance}_{\text{del}}\backslash\text{Speed B}_{\text{del}}$	Low	Medium	High
CC Guided	Dist	Dist	Dist
DC Target	DC_Low	DC_Med	DC_High
Too Close 1/1	DC_Low	DC_Low	DC_Med
Too Close 1/2	DC_Low	DC_Low	DC_Low

3.6.3 Second Mapping to a Quantitative Model

After having successfully adapted the initial qualitative model, we started another iteration by defining our second mapping to a quantitative model. More precisely, we mapped the model with the new *Too Close 1/2* state defined in the FSM in Figure 3.10 with its transition conditions defined in Table 3.5 as well as the adapted DC FSM transition conditions in Table 3.6 to a quantitative model with the values given in Table 3.7. This was also done using the CEGAR refinement that the workflow adopts, since it is also defined for the concrete model. For adopting the mapping, we used the information gathered during the workflow-step of checking the counterexample. Basically, we split the range associated with *Too Close* into two regions, one covering all distances from which a $distance <= 0\,m$ is reachable and one covering the rest of the range.

Note, that the distance of *DC Target* has not been increased in the course of this change of the quantitative model [60].

Table 3.7: Summary of mapping found to be valid for specific speed values [60, Table 7]

	Value/Range
High Speed	$30\,m/s$
Medium Speed	$22.5\,m/s$
Low Speed	$15\,m/s$
Time-step	$1\,s$
CC Guided	$(35\,m,\ \infty]$
DC Target	$(20\,m,\ 35\,m]$
Too Close 1/1	$(7.5\,m,\ 20\,m]$
Too Close 1/2	$(0\,m,\ 7.5\,m]$
Collision	$[-\infty,\ 0\,m]$

At first, we were tempted to consider this model *validated* because the values appear realistic in terms of physics; particularly the distances and accelerations ($-7.5\,m/s^2$, $0\,m/s^2$, and $+7.5\,m/s^2$). And, because the abstract qualitative model was an *over-approximation* of the concrete quantitative model, the workflow would have already been completed successfully [60].

However, the model still has a severe validation problem when looking at the speed values. It is unrealistic to allow only selected concrete speed values, since vehicles may also drive speeds between those values.

3.6.4 Introducing Speed Ranges

To overcome this shortcoming, each speed class in the qualitative model was mapped to a range of speed values as follows:

- High Speed = $(25\,m/s,\, 30\,m/s]$
- Medium Speed = $(20\,m/s,\, 25\,m/s]$
- Low Speed = $(15\,m/s,\, 20\,m/s]$

Unfortunately, when each vehicle randomly picks one speed value from the same speed range, holding the distance is no longer guaranteed. Hence, the abstract qualitative model is no longer an over-approximation of the quantitative model since it does not have the corresponding transitions. Hence, the qualitative model has to be modified again[60].

Table 3.8: Transition conditions according to speed ranges [60, Table 8]

Speed A\B	Low	Medium	High
Low	H/I/D	H/I	H/I
Medium	H/D	H/I/D	H/I
High	H/D	H/D	H/I/D

H . . . Hold distance (Dist_H)
I . . . Increase distance (Dist_I)
D . . . Decrease distance (Dist_D)

The adapted transition conditions in Table 3.8 reflect the fact that the distance is no longer guaranteed to be kept when both vehicles drive at the same qualitative speed by altering the main diagonal of the table.

More detailed analysis also revealed that the entries for Vehicle A and Vehicle B driving *Low Speed* and *High Speed*, respectively, and the other way around, have to be altered as well. We assume that Vehicle A drives *Low Speed* and Vehicle B drives *High Speed*. According to the defined speed ranges, Vehicle A and Vehicle B may drive $20\,m/s$ and $26\,m/s$, respectively. This accounts for a speed difference of $6\,m/s$ and, therefore, an increase of the distance of $6\,m$ in one time-step. Looking at Table 3.7 with this information, one notices that for each (qualitative) distance class, one may find two distances x and $y = x + 6\,m$ that are both in the defined range of that class. This means that the qualitative distance may be kept although Vehicle B drives *High Speed* and Vehicle A drives *Low Speed*. The same goes for Vehicle B driving *Low Speed* and Vehicle A driving *High Speed*.

Model-checking with these adaptations generated a counterexample, i.e., collision is possible, even with the previously adapted Distance Classification (with *Too Close 1/1* and *Too Close 1/2*) as shown in Figure 3.10.

In retrospect, it is evident that the qualitative model causes such a counterexample because it considers the possibility that Vehicle A drives faster than Vehicle B, although both vehicles drive at *Low Speed*. This (slightly) faster speed of Vehicle A leads to a decrease in the distance and finally to a collision.

3.6.5 Analysis and Another Model Modification

Since checking the counterexample revealed that it is real, we performed a detailed analysis of the qualitative model.

This analysis revealed that, whenever Vehicle A is in the *Too Close 1/2* state, its speed is *Low Speed*. The critical case occurs when Vehicle B also drives *Low Speed*. Since we abstract the exact speed values of both vehicles, we do not know whether Vehicle A is approaching or not. The transition conditions of this qualitative model take that into account by allowing the transition from *Too Close 1/2* to *Collision*.

In order to solve this problem, we changed the conditions according to Table 3.9. This adaptation of the qualitative model reflects a mapping of *Low Speed* to exactly one concrete speed value, so that, when both vehicles have this speed, the distance can only stay the same. Hence, there are fewer transitions again, and model-checking with this adaptation did not lead to a counterexample.

Table 3.9: Transition conditions when *Low Speed* is mapped to exactly one concrete value [60, Table 9]

Speed A\B	Low	Medium	High
Low	H	H/I	H/I
Medium	H/D	H/I/D	H/I
High	H/D	H/D	H/I/D

H . . . Hold distance (Dist_H)
I . . . Increase distance (Dist_I)
D . . . Decrease distance (Dist_D)

3.6.6 Modification of the Mapping

As indicated above, keeping the distance if both vehicles drive at the same qualitative speed, e.g., *Low Speed*, is only guaranteed if the value is mapped to a single concrete speed value.

Hence, we modified the mapping to take this latest change of the qualitative model into account:

- High Speed = $(22.5\,m/s, 30\,m/s]$
- Medium Speed = $(15\,m/s, 22.5\,m/s]$
- Low Speed = $15\,m/s$

According to the workflow, the question had to be answered, whether this model can still be *validated*. The changes of the speed ranges and, at the same time, keeping the time-step at $1\,s$ have increased the possible accelerations in this model in the range $(-15\,m/s^2, +15\,m/s^2)$. At least for current automotive vehicles, we judged them as unrealistic [60].

To solve the issue of high acceleration values, we changed the mapping so that the overall speed range is divided into four instead of three sub-ranges:

- High Speed = $(25\,m/s, 30\,m/s]$
- Medium Speed = $(20\,m/s, 25\,m/s]$
- Low Speed = $(15\,m/s, 20\,m/s]$
- AbsLow Speed = $15\,m/s$

Whereby the new speed class of *AbsLow Speed* corresponds to an absolute low speed, which cannot be dropped below.

With the new mapping, we consider the quantitative model validated. However, the qualitative model is clearly not an over-approximation since it does not yet have the fourth speed class *AbsLow Speed*. Therefore, we had to include *AbsLow Speed* in the qualitative model.

3.6.7 Adaptation of the Qualitative Model

Due to the introduction of the new speed class *AbsLow Speed* via the mapping, all FSMs had to be updated. Of course, the transition conditions had to be updated, too.

First and foremost, we updated the transition conditions of the Distance Classification. The new conditions are given in Table 3.10.

Table 3.10: Transition conditions with *AbsLow Speed* [60, Table 12]

Speed A\B	AbsLow	Low	Medium	High
AbsLow	H	H/I	H/I	H/I
Low	H/D	H/I/D	H/I	H/I
Medium	H/D	H/D	H/I/D	H/I
High	H/D	H/D	H/D	H/I/D

H . . . Hold distance (Dist_H)
I . . . Increase distance (Dist_I)
D . . . Decrease distance (Dist_D)

Since introducing a new speed class means that all entities need to handle this new class, all other FSMs needed an update too. However, showing all of the updated models as well would cause more confusion than contribute to understanding at this point. Therefore, only the FSM of DC and their transitions conditions are given in Figure 3.11 and Table 3.11, respectively. The FSMs of CC, Coordinator and Vehicle B are extended analogously.

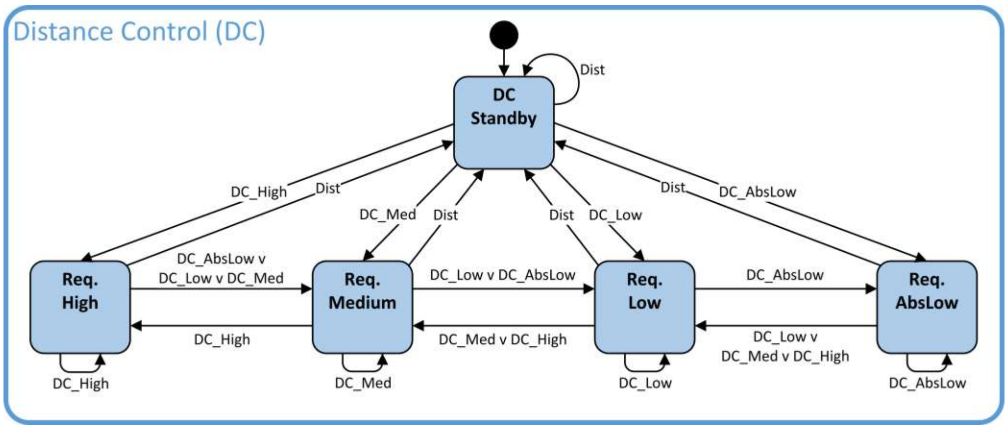


Figure 3.11: FSM of Feature DC including the new speed class *AbsLow Speed*

Table 3.11: Transition conditions related to the DC FSM when the Distance Classification has two *Too Close* states and four speed classes are used (extended version of [60, Table 11])

$\text{Distance}_{\text{del}} \backslash \text{Speed B}_{\text{del}}$	AbsLow	Low	Medium	High
CC Guided	Dist	Dist	Dist	Dist
DC Target	DC_AbsLow	DC_Low	DC_Med	DC_High
Too Close 1/1	DC_AbsLow	DC_AbsLow	DC_Low	DC_Med
Too Close 1/2	DC_AbsLow	DC_AbsLow	DC_AbsLow	DC_Low

Model-checking this qualitative model shows that a collision can occur. We did not expect that, since according to our intuition introducing an additional speed class into the abstract model reduces the approximation error. However, it entails the need for a more detailed Distance Classification as well. Keep also in mind, that the distance can change in this model, although the speed classes are the same (except for the case of both vehicles driving *AbsLow Speed*). The counterexample is given as follows:

1. Vehicle B drives *High Speed* and Vehicle A drives *High Speed* → Target changes from *CC Guided* to *DC Target*.

2. Vehicle B decelerates to *Medium Speed* and Vehicle A drives *High Speed* (DC sees *High Speed* of Vehicle B from the previous step) → Distance changes to *Too Close 1/1*.

3. Vehicle B decelerates to *Low Speed* and Vehicle A decelerates to *Medium Speed* → Distance changes to *Collision*, since the difference in the vehicle speeds allows “jumping over” the *Too Close 1/2* state.

However, this is a spurious counterexample due to the abstraction.

Table 3.12: Transition conditions related to the DC FSM when the Distance Classification has two *Too Close* states and four speed classes are used (altered version of [60, Table 13])

Distance$_{\text{del}}$\Speed B$_{\text{del}}$	AbsLow	Low	Medium	High
CC Guided	Dist	Dist	Dist	Dist
DC Target	DC_AbsLow	DC_Low	DC_Med	DC_High
Too Close 1/1/x	DC_AbsLow	DC_AbsLow	DC_Low	DC_Med
Too Close 1/2	DC_AbsLow	DC_AbsLow	DC_AbsLow	DC_Low
Too Close 2	DC_AbsLow	DC_AbsLow	DC_AbsLow	DC_AbsLow

3.6.8 Modification of the Qualitative Model and the Mapping

According to CEGAR refinement, the state *Too Close 1/1* has to be split to fix the spurious counterexample. Analogously to fixing the previous spurious counterexample (in Subsections 3.6.2 and 3.6.3), this was done by first analyzing the counterexample in terms of physics and then splitting the state into two regions. The process resulted in *Too Close 1/1/1* (10 m, 20 m] and *Too Close 1/1/2* (7.5 m, 10 m]. Due to analogy with the extension of the Distance Classification from one *Too Close* state (Figure 3.6) to the one containing *Too Close 1/1* and *Too Close 1/2* (Figure 3.10), we omit the concrete FSM and their corresponding tables at this point.

This more detailed Distance Classification fixed the occurrence of this spurious counterexample. However, model-checking revealed another counterexample, a real one [60]. Therefore, we fixed the counterexample by inserting the new distance class *Too Close 2*. Not that this time we consider the distance *DC Target* to increase, as in contrast to "just" fixing a spurious counterexample.

This insertion of a new state into the Distance Classification requires taking it into account in the transition conditions of DC as well, see Table 3.12

Model-checking this model did not reveal any counterexample. Hence, we had to determine its mapping to a quantitative model. The result of this mapping step is given in Table 3.13.

Table 3.13: Summary of mapping found to be valid for speed ranges ([60, Table 14])

	Range
CC Guided	(50 m, ∞]
DC Target	(35 m, 50 m]
Too Close 1/1/1	(25 m, 35 m]
Too Close 1/1/2	(22.5 m, 25 m]
Too Close 1/2	(15 m, 22.5 m]
Too Close 2	(0 m, 15 m]
Collision	[$-\infty$, 0 m]

This quantitative model also passed our validation, but the check for over-approximation revealed that the abstract model had to be modified again to cover the whole behavior of this concrete model. Model-checking this modified abstract model did not reveal any counterexample.

Determining the new mapping and validation of the resulting concrete model were straightforward since the latest modification of the qualitative model did not involve splitting states. Instead, only transitions between already existing states were added or changed. Finally, the abstract qualitative model was an over-approximation of the concrete quantitative one. Therefore, the workflow finished successfully.

Determining the new mapping and validation of the resulting concrete model were straightforward since the latest modification of the qualitative model did not involve splitting states. Instead, only transitions between already existing states were added or changed. Finally, the abstract qualitative model was an over-approximation of the concrete quantitative one. Therefore, the workflow finished successfully.

In terms of verification, this means that the concrete quantitative model has been shown indirectly to not run into any collision, since the abstract qualitative model has been formally model-checked against this condition, and it is an over-approximation of the concrete quantitative model[60].

The spotlight of this chapter was on changes that were relevant to fix (spurious and real) counterexamples. However, especially during the checking for over-approximation, one identifies missing transitions between states. In our case, we identified a whole set of additional transitions in order to grantee over-approximation. However, those transitions are never taken because Vehicle A adapts its speed in such a way that the corresponding conditions never become true. For completeness, the entire final qualitative model is given in Section A.1 of the Appendix.

3.7 Related Work

In this section we give some related work to CEGAR and feature interaction[1].

Since model checking intrinsically faces combinatorial explosion, abstraction techniques were studied, e.g., in [31, 32, 55], to reduce the state-space. These techniques provide a systematic way to generate an abstract model from a concrete one with one-sided error. In contrast to the approach used here, this work considers the concrete model as given and fixed. Hence, these techniques start from a given concrete model, while our workflow starts from an abstract model either given or defined in due course. Still, these abstraction techniques can be used in the course of enacting the workflow.

Clarke et al. [31, 32] distinguish in CEGAR between *spurious* and real counterexamples, where the former only occur in the abstract model but not in the concrete one, while the latter occur in both. Given a spurious counterexample, CEGAR provides a systematic and automated way to refine the abstract model. If the counterexample is real, the algorithm stops and states that the system does not fulfill the property checked. If no

[1]This section is a combination of parts directly taken from [60, 53, 71].

counterexample can be found, however, the concrete system is considered safe. Either way, the technique can be used to make a statement whether a concrete model fulfils a given property, by model-checking it against abstract models derived from the concrete one. In contrast, the goal of the workflow is to systematically construct a concrete model that fulfils the property. Since the design starts at the abstract level, also real counterexamples are used in the workflow to fix the abstract model. In effect, all counterexamples are used as sources of information for (manually) creating a concrete and quantitative model.

Wang et al. [82] and Tian et al. [81] proposed meanwhile technical improvements on CEGAR with regard to the detection of spurious counterexamples and the refinement of the model. For our case study, we employed the original techniques (manually). For automating, it could be interesting to apply these improvements.

The seminal CEGAR approach has been applied to different verification tasks for already given systems. Nellen et al. [64, 65] used CEGAR to verify an already given programmable logic controller (PLC) against safety properties, proposing two approaches for that. Stursberg et al. [80] built on CEGAR for verification of a cruise control system using counterexample-guided *search*. Hybrid automata representing an already given ACC implementation are used for its verification in a closed-loop setting. In addition to the integrated verification (based on the same theory), our design workflow includes steps for constructively designing such a CPS in the first place, including, e.g., the determination of system parameters.

Clarke et al. [29] extended CEGAR for verification of *hybrid systems*. While we derived a quantitative model from a qualitative one in the course of our case study, it is still a finite system since we restrained it, e.g., to a finite number of distinct speed values. Hence, we did not have to use these extended techniques for hybrid systems. Still, applying them should be possible in the context of our workflow.

While all this work building on CEGAR is dedicated to verification using model checking, Abrial et al. [15, 16] illustrated with numerous examples Event-B, an approach to formalizing hybrid systems. Event-B includes both modeling and formal reasoning using formulas. A system of formulas for modeling a hybrid system is developed strictly incrementally and verified in due course. However, we could neither find a description of how Event-B generally works nor a defined workflow for its application in [15, 16]. Our workflow also integrates verification into the systematic development of a quantitative model, but it starts with a qualitative model. The workflow takes into account that a model on a higher level of abstraction may have problems and allows, therefore, to iterate between levels of abstraction for fixing them.

Software and CPS design in practice integrates verification as well, of course, both informally and formally. However, we are not aware of any previous approach that would systematically determine changes of models on different levels of abstraction based on verification results like our workflow.

FIs may result directly from conflicting requests to a single variable (as studied for speed before in the context of model checking in [52]). This is sufficient for detecting that there is an FI, but not for investigating its influence on the physical system. We study this influence on the distance to another vehicle, which is important for model checking the features CC and DC together, as well as their coordination.

The formulation of resolutions as proposed in [24] uses the Situation Calculus implemented in GOLOG [57]. This work did not investigate, however, the physical effects from a feature interaction of a software variable, and it did not include independent physical variables outside the control of the software. Hence, no verification was possible of whether such a resolution as specified actually achieves its purpose.

According to [74], in our model only longitudinal motion is possible. The steering angle for overtaking maneuvers, for example, would require a more elaborate model, but this is outside the scope of our investigations.

3.8 Discussion

It cannot be ensured that the workflow does come to an end. Among other things, it is possible that the initial qualitative model is so wrong that it is impossible to find a valid quantitative model by mapping the qualitative states to concrete values (or ranges). However, we consider this unlikely when domain experts create the initial qualitative models.

The result of the workflow heavily depends on the initial qualitative model, generated counterexamples, and the decisions made during mapping the qualitative states to concrete values (or ranges). Since each of these decisions and each refinement made due to a spurious counterexample stack on each other, the influence on the final result of each individual decision made while performing the workflow is difficult to predict. Thus, generating a solution that can be considered optimal is very unlikely.

During the case study, all steps of the workflow were done manually. Clearly, this is not feasible for larger systems. However, some steps may be automated, e.g., revising the qualitative model after a spurious counterexample was found. Others may become tool supported, e.g., checking the over-approximation. And again, others may not be automatable, e.g., revising the design after a real counterexample was found. Which parts of the workflow can be automated may also depend on the domain to which the workflow is applied. Therefore, the scalability of this approach has still to be shown.

CHAPTER 4

Model-Checking of Process Consistency

One way to look at cyber-physical systems (CPS) is in a process-oriented way. This view allows verification of a CPS in a specific context (its environment) by modeling the behavior of the CPS as processes to be performed by the CPS. In turn, its environment can be seen as a composition of objects that are manipulated by the process. It even allows modeling the communication between different CPSs by utilizing the concepts of interprocess communication and synchronization. Therefore, in this chapter, we see the behavior of CPSs as processes and the environment as the context those processes are performed in.

This chapter starts by giving some additional background information on object life cycles (model of the environment objects based on UML State Machines) and semantic action specification used to connect process models (UML Activities) with object life cycles. After that, we present a verification workflow for verifying process-oriented models using a model-checker. Following, we discuss how our workflow uses high-order processes to orchestrate processes. Subsequently, a meta-model for specifying the connection between processes and their environment is defined. The following section defines a UML Profile for annotating UML Activities and shows how annotation enables the reuse of existing process models in different contexts (environments). Since the verification is done with the model-checker nuXmv [27], the follow-up section goes into details of the translation from UML Activity diagrams into the input language of the model-checker. Then we define what we mean by consistency and how those properties can be expressed in terms of *linear time logic* (LTL). In the following section, we then use this workflow to provide evidence that the approach is applicable to CPS. The chapter concludes with related work and a short discussion of the results.

4.1 Background

This background section gives some background information on the concept of Object Life Cycles (OLC) that is used to model the behavior of objects. It also gives background information on semantic action specification that is later extended and used to connect process models with OLCs.

4.1.1 Object Life Cycle

An object life cycle (OLCs) is a way to define the behavior of an object by specifying legal sequences of states and activities [75]. It is only of secondary importance which type of object is involved. Thus, OLCs are used for the modeling business objects [48, 37, 73], objects in object-oriented database schema [68], parts of a cyber-physical systems and so forth. However, all "types" of OLCs have in common that they specify states of objects and possible transitions between those states where the transitions are triggered by some activity performed on the object.

Object life cycles are often seen as passive entities that describe the states of the objects they are representing. Thus, they never perform some sort of activity (on other objects). In this thesis, OLCs are seen differently. OLCs may influence each other by triggering behavior of each other, like, e.g., physical objects influencing each other's behavior in the real world. Thus, we use extended OLCs, defined by Hoch et al. [47], which allow modeling interactions between physical objects.

In this thesis, we use finite state machines to define OLCs, since they perfectly fit when it comes to describing behavior in terms of states and transitions and are also part of the UML standard like the activity diagrams we used for modeling the processes.

Figures 4.1 and 4.2 shows the OLCs associated with our running example, the "payment handling" process. Each OLC is directly associated with one of the processes. It is almost a philosophical question whether the two invoices are different entities, one for each company, or if they are the same invoice, but each company has its unique view on then it comes to the state of the invoice. However, for the remainder of the thesis, we consider them as different views of the same object. Therefore, the state machine *DCP Invoice* is the view of the delivering company and the *CCP Invoice* is the view of the customer company on the same invoice.

Each OLC is modeled using several *Regions*. One region – the top one – specifies the possible transitions an external entity (process model or another OLC) may trigger. For example, the transition from *DCP Initial* to *Set Created* that is triggered via *DCP_IC*. In this example, all other regions contain state machines that only react to the state of the former state machine and represent a particular attribute of the object modeled by the OLC, e.g., the object being created.

In addition to the state machine, each OLC defines *Behaviors* that each trigger some transition in the state machine. For the OLCs of the "payment handling" process, we defined seven behaviors – four for *DCP Invoice* and three for *CCP Invoice* – which are given in the class diagram given in Figure 4.3.

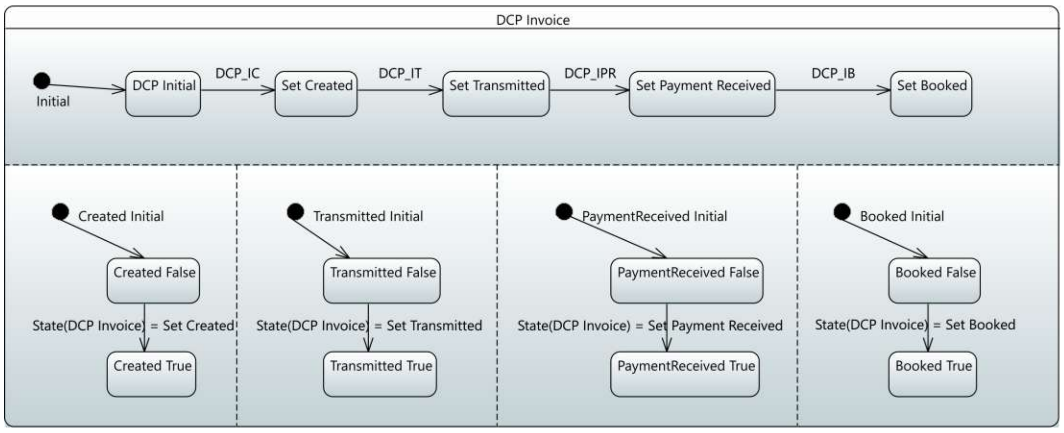


Figure 4.1: Object life cycle associated with *Delivering Company Process* given in Figure 2.2

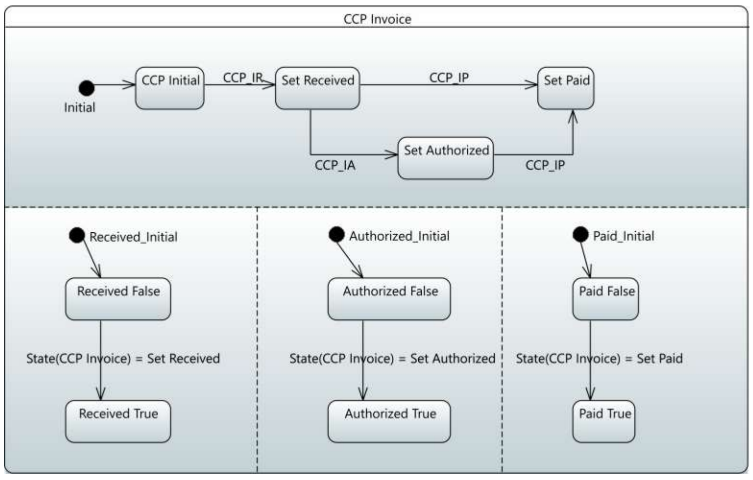


Figure 4.2: Object life cycle associated with the *Costumer Company Process* given in Figure 2.2

Each behavior triggers a transition in the OLC's state machine. For example, the behavior *ReceiveInvoice* triggers – via the signal *CCP_IR* – the transition from state *CCP Initial* to state *Set Received* of the *CCP Invoice* state machine. In this particular case, even another transition is triggered. Since the top-region of the *CCP Invoice* ends up in the state *Set Received* a changeover from state *Received False* to *Received True* in the bottom-left region is triggered. Thus, setting the attribute *received* of the OLC.

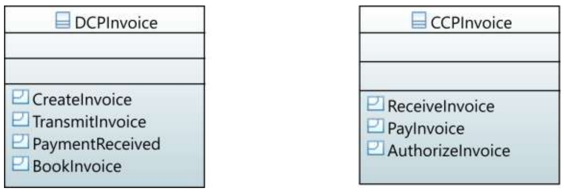


Figure 4.3: Classes defining the Behaviors that can be executed on the DCP Invoice and CCP Invoice object life cycle

4.1.2 Semantic Action Specification

The *semantic action specification* defines an attachment of pre- and postconditions, grounded in OLCs, to an action of an activity [47].

To illustrate how the *semantic action specification* works, we use the running example of the action *Receive Invoice* of the *Receive Invoice* of the *Costumer Company Process*.

Listing 4.1: Semantic Action Specification of Receive Invoice Action (adopted version of Listing 1 in [47])

```
Receive Invoice:
   Pre: --
   Post: received(CCPInvoice)
```

Listing 4.1 gives the concrete semantic specification attached to *Receive Invoice*. In it, the precondition (Pre) is set to be empty, which means that the precondition is always met and, therefore, the action is activated as soon as a control flow token is provided to it. However, in general, an action's execution is prevented until its precondition is met. In contrast, the postcondition (Post) specifies the condition upon fulfillment of which the execution of the action is considered successful. In our concrete example, the execution of the action *Receive Invoice* is considered successful when the attribute *received* of the OLC *CCP Invoice* is set.

An extended version of the *semantic action specification* plays a major role in the proposed verification workflow since it connects actions with OLCs.

4.2 Proposed Verification Workflow

This section gives an overview of the proposed workflow used to verify cyber-physical systems. For details of individual steps, please refer to the relevant sections.

The goal of this workflow was to allow for verification of process models. At the same time, we attempted to provide a workflow that allows the modeling of process models and object life cycles (OLC) as independently as possible. We think this is an important feature of our approach since this allows reusing existing process models as well as OLCs.

The proposed workflow given in Figure 4.4 has in total 5 steps (excluding the creation of the input files and models). Each step relies on different input files and/or models partly generated by the preceding step.

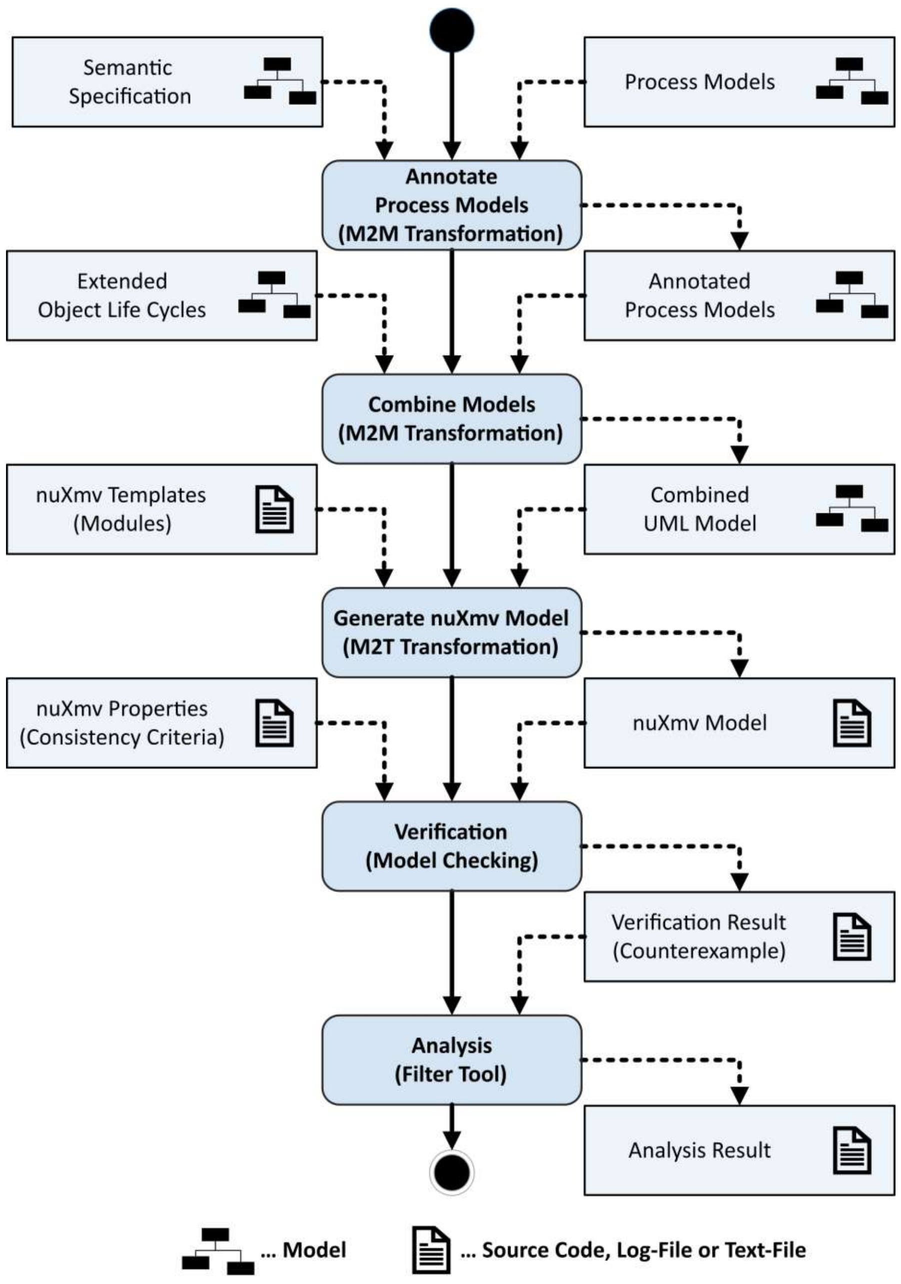

Semantic Specification
Process Models
Annotate Process Models (M2M Transformation)
Extended Object Life Cycles
Annotated Process Models
Combine Models (M2M Transformation)
nuXmv Templates (Modules)
Combined UML Model
Generate nuXmv Model (M2T Transformation)
nuXmv Properties (Consistency Criteria)
nuXmv Model
Verification (Model Checking)
Verification Result (Counterexample)
Analysis (Filter Tool)
Analysis Result
... Model
... Source Code, Log-File or Text-File

Figure 4.4: Workflow of Verification Approach

The first step of the workflow has at least two input models, the *Process Models* that actually model the processes we want to verify as a UML Activity diagrams and the *Semantic Specification*, a model containing the semantic specification of certain elements of the processes. This step takes the process models and uses a UML Profile, created for this purpose, to annotate the activity diagrams. The output of this step are the *Annotated*

Process Models. The main difference between *Process Models* and the annotated ones is that the former are independent of object life cycles (OLCs) and the latter are already "bound" to specific OLCs.

The next step, *Combine Models* just does this. It combines the files containing the *Extended Object Life Cycles* and the *Annotated Process Models* into one file containing all models and, therefore, all information about the processes and their context.

The combined model-file (*Combined UML Model*) is then used together with predefined code templates (*nuXmv Templates*) to generate code that represents an *nuXmv Model*. During the model-to-code transformation performed in this step, an implicit transformation of UML Activity diagrams to multiple synchronized Finite State Machines (FSMs) is done.

In the Verification step, the nuXmv Model is verified against the provided set of LTL properties. This set contains predefined and optionally user-defined properties. The predefined properties are used for verifying "basic" consistency between processes and OLCs. User-defined properties may contain additional constraints on the execution of the processes and the OLCs. However, the model checker nuXmv takes those properties and verifies each separately. In the case of the successful verification against a specific property, nuXmv states that the property evaluates to True (or in the case of bounded model checking terminates at the given maximal bound). In the case of a failure, a counterexample leading to the violation of the property is generated. For documentation of the result and to enable analysis, later on, the output of the model-checker is written to a log-file (*Verification Result*).

The last step of the verification workflow is the analysis of the *Verification Results*. Since the model is quite complex we found that some tool is needed to analyze the generated counterexamples. The tool parses the log-file that the model-checker generated (*Verification Result*) and filters the trace according to a given set of strings. Variables occurring in the trace are only kept if some string of the set is part of the name of the variable. E.g., the String ".ACTIVE" matches the variable *mainAct.Call_Car_Process.InitialNode.ACTIVE*. The filtered trace is easier to read, and therefore, it is easier to spot the cause of the counterexample. In the standard configuration of the tool, it uses a predefined set of strings that have shown to be valuable to spot where a process came to a halt or why an OLC has entered an error state.

The entire workflow is tool-supported. Doing this workflow by hand is not feasible. As the main platform, we chose the Eclipse Modeling Framework (EMF) since it allows the integration of many plugins needed for modeling, model-to-model (M2M) transformation and model-to-code transformation.

For creating the necessary UML models, Papyrus is used since it is integrated well into the Eclipse Modeling Framework and also provides a graphical way for defining UML Profiles. In theory, the workflow is not limited to Papyrus models. However, practice has shown that different UML modeling tools generate slightly different models even though the diagrams look similar.

For defining the semantic specification, we implemented an EMF-based editor. This editor allows connecting process models and OLCs by giving means to define semantic constraints and attach them to actions and control flows. The model generated by it then represents the input *Semantic Specification* of the model annotation process.

The two M2M transformations are implemented using QVTo (Query View Transformation Operational) rather than, e.g., ATL (Atlas Transformation Language), since it integrates well with the EMF Meta-Models as well as the UML Profiles generated with Papyrus. In addition, QVTo allows using APIs and integrating source code written in other programming languages, e.g., Java, whenever the capabilities of the QVTo are insufficient.

We rely on a Model-to-Text approach for code generation of the nuXmv source code. The implementation is based on the JAVA dialect Xtend [14], a statically typed template language, which allows modularization and also integrates well with other EMF tools.

For the actual verification, the model-checker nuXmv is used. We are aware that we could have used another model-checker like SPIN [79] or PRISM [11]. nuXmv was partly chosen for historical reasons because our early work focused on verification of finite state machines only and later evolved towards verification of processes.

The Filter Tool for analysis of the log-file was implemented in Python. Starting as a little tool for analyzing the output of the model-checker, mainly for validating results, it became an essential part of the workflow.

The individual steps of the workflow are coordinated with each other via the Modeling Workflow Engine 2 (MWE2), which starts each step, passes the inputs to the corresponding tools, and stores the generated outputs.

Taken together, this provides a tool-supported workflow for the verification of processes that also facilitates the reuse of existing models.

4.3 Synchronization of Processes

For reasoning whether a set of processes is consistent with each other and the involved (business) objects, synchronization of processes is quite important. One approach could have just assumed that all processes run in parallel. However, to provide more flexibility, the synchronization is modeled explicitly using an Activity diagram to represent a higher-order process that orchestrates all other processes. In this way, it is possible to define complex scenarios, including e.g., conditional or cyclic execution of processes.

To reflect that *Customer Company Process* and *Delivering Company Process* of our running example “payment handling” run in parallel we define the UML Activity given in Figure 4.5. In the diagram, we use the *Fork Node* to split up the initial control-flow into two parallel flows and then call the processes via *Call Behavior Action* referencing the corresponding *Activities*. After each of the processes has finished, the control-flow is joined back together, and the higher-order process ends.

This higher-order synchronization via control-flows differs significantly from the synchronization done via *Send Action* and *Receive Event Action* where asynchronous inter-process communication occurs. The former allows to synchronize the beginning of the execution of processes, whereas the latter allows synchronization during the execution of processes.

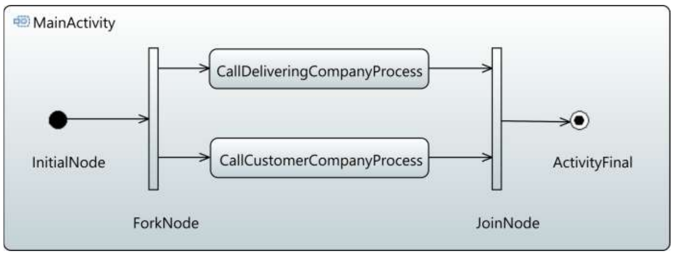


Figure 4.5: Scheduling “payment handling” process

4.4 Extended Semantic Action Specification

In previous work [47], we defined the specification only consisting of post- and precondition. In this thesis, an extended version of the semantic action specification is used, which also includes the behavior of the OLC that the action is supposed to trigger.

Hoch at al. [47] uses *goal regression* to determine the corresponding behavior of a postcondition. Since goal regression on complex OLCs may not be easy and it is out of scope of this thesis, the behavior is modeled explicitly. However, goal regression could still be used to determine the behavior.

Listing 4.2 shows an informal representation of the extended semantic action specification attached to the *Receive Invoice* action of the *Costumer Company Process* given in Figure 2.2. In addition to pre- and postcondition, it defines the behavior to be triggered by the action as *ReceiveInvoice* of the OLC *CCP Invoice*.

Listing 4.2: Extended Semantic Action Specification of Receive Invoice Action (compare Listing 4.1)

```
Receive Invoice:
   Pre: --
   Post: received(CCP Invoice)
   Beh: call(ReceiveInvoice, CCP Invoice)
```

4.5 Modeling Semantic Specifications

The semantic specification logically connects entire process models with OLCs and vice versa by allowing the definition of multiple extended action specifications and semantic specifications attached to control flows.

For modeling entire semantic specifications, the meta-model given in Figures 4.6 and 4.7 was defined. Instances of the meta-class *Specification* represent the semantic specification, which is composed of instances of other meta-classes.

Each *Specification* has the attributes *name* and *id* to (uniquely) identifying it. Via the attribute *models* it also allows referencing *UML Models* for specifying the process models (*UML Activities*) and OLCs (*UML State Machines*) to be connected and, therefore, defining the context of the specification. By specifying the models the semantic specification is intended to connect, it is possible to limit the choices provided by the EMF editor

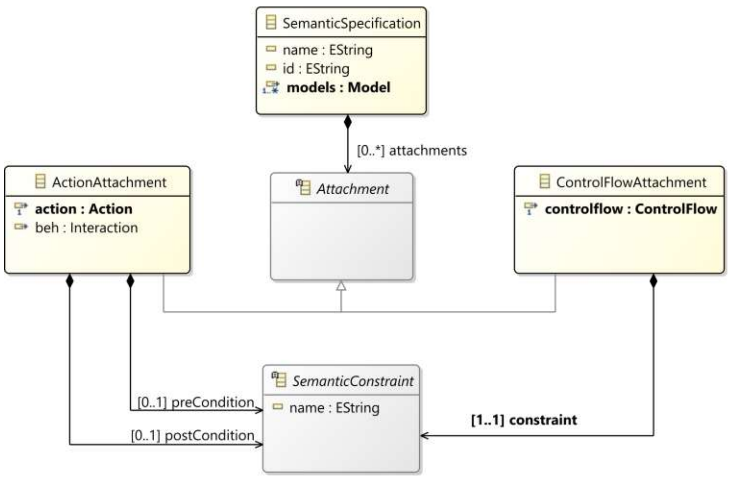


Figure 4.6: Meta-model for defining the semantic specification used to connect processes (Activity diagrams) and OLCs – Part 1 of 2

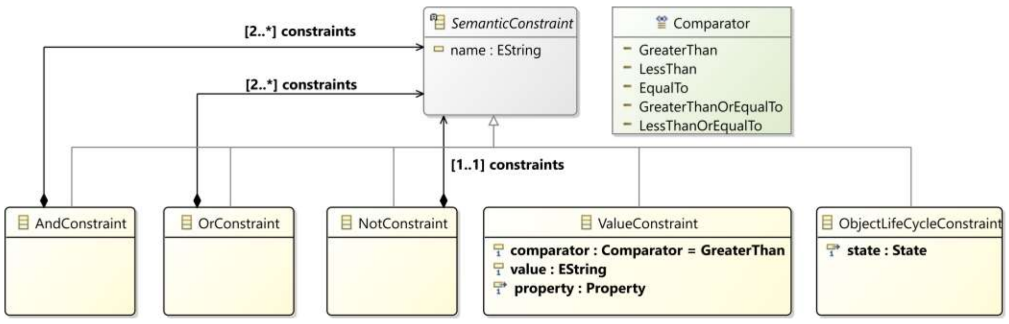


Figure 4.7: Meta-model for defining the semantic specification used to connect processes (Activity diagrams) and OLCs – Part 2 of 2

when it comes to picking an action or behavior. This has been shown beneficial since the EMF typically gives all elements of all imported files that fit the required type. Since it is not unusual to have multiple models in one file, the user may be given a long list of elements. By filtering this list by the model they are contained in, this list is reduced to the relevant ones. Giving the models explicitly also enables the possibility to check whether all semantic constraints, defined in the semantic specification, only reference intended models.

The actual connection between process models and OLCs is modeled via the instances of the meta-class *Attachment* the *Specification* holds in its attribute *attachments*. We defined two types of attachments, one for attaching an extended semantic action specification to an action – *Action Attachment* – and one for attaching a semantic constraint to a control flow – *Control Flow Attachment*. The former has the attributes *preCondition* and *postCondition* for holding semantic constraints defining the precondition and postcondition, respectively, of the extended semantic action specification. It also allows referencing an OLC behavior – attribute *beh* – as well as the action – attribute *action* – the extended semantic action specification is attached to. A *Control Flow Attachment* only holds a single semantic constraint – attribute *constraint* – and has a reference to the control flow the constraint is attached to. Together, they allow attaching constraints to actions and control flows of a UML Activity.

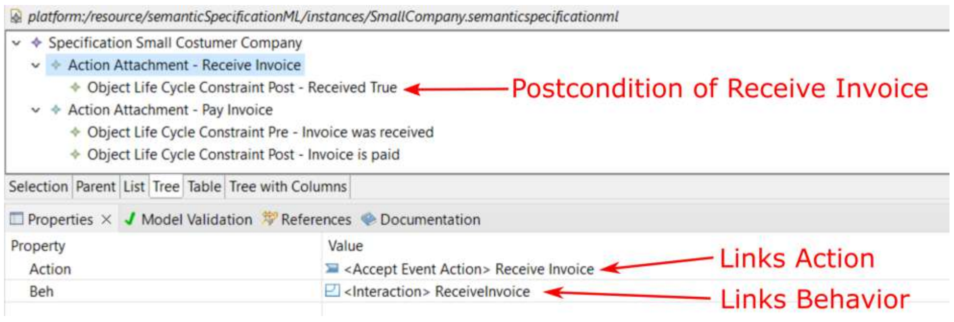
For the definition of semantic constraints, instances of the meta-class *Semantic Constraints* are used. Figure 4.7 shows this meta-class and its relation to its sub-meta-classes. These classes can be used to form more or less complex semantic constraints. *And Constraint* and *Or Constraint* combine the contained semantic constraints with logical AND and OR, respectively. A *Not Constraint* simply negates the contained semantic constraint. The actual connection between process models and OLCs is established via instances of the meta-class *Value Constraint* and *Object Life Cycle Constraint*.

- *Value Constraint* ... is a constraint that checks whether the value of a OLC property holds a specific condition. Besides the name inherited from *Semantic Constraints* a *Value Constraint* has three attributes. Attribute *property* gives the property that is looked at during the evaluation of the constraint. The value of the attribute *comparator* is an instance of the meta-class *Comparator* and decides which Boolean-function is used during compare. The attribute *value* holds the constant that the property is compared to.

- *Object Life Cycle Constraint* ... is a constraint that checks whether the OLC is in a specific state.

Value Constraint and *Object Life Cycle Constraint* are implicitly connected to a specific OLC via attributes *property* and *state*, respectively.

Figure 4.8 shows the semantic specification of the *Small Costumer Company* process. Compared to the *Customer Company* process given in Figure 2.2 this process consists only of the actions *Receive Invoice* and *Pay Invoice*.


platform:/resource/semanticSpecificationML/instances/SmallCompany.semanticspecificationml

- Specification Small Costumer Company
  - Action Attachment - Receive Invoice
    - Object Life Cycle Constraint Post - Received True
  - Action Attachment - Pay Invoice
    - Object Life Cycle Constraint Pre - Invoice was received
    - Object Life Cycle Constraint Post - Invoice is paid

Selection | Parent | List | Tree | Table | Tree with Columns

Properties | Model Validation | References | Documentation

Property	Value
Action	<Accept Event Action> Receive Invoice
Beh	<Interaction> ReceiveInvoice

Figure 4.8: Semantic specification of actions used in the Small Costumer Company process

In the upper half of the figure the tree structure of the semantic specification is given. For simpler reading the editor labels each *Action Attachment* (and *Control Flow*

Attachment) with the name of the meta-class as well as the action (control-flow) the semantic constraints are attached to. With this knowledge, just looking at the highlighted blue tree-element we already know that this *Action Attachment* expresses the extended semantic action specification of the action *Receive Invoice*. Details on the attachment are given in the bottom half of the figure. Here we can see the action, again, as well as the OLC *Interaction – ReceiveInvoice* – which is defined as behavior (Beh).

This meta-model allows defining complex semantic constraints in a tree-like structure by nesting semantic constraints. As an example of such a structure Figure 4.9 shows the precondition of the action *Pay Invoice*, which defines that invoices with $amount \geq 150$ need to be authorized and invoices with an $amount < 150$ are not allowed to be authorized. In both cases, the invoice must be received as well. To specify the condition, an *Or Constraint* is used as the “root” of the tree-like structure. This *Or Constraint* then contains two *And Constraints*, each of which containing three *Semantic Constraints* actually connecting to the OLC.

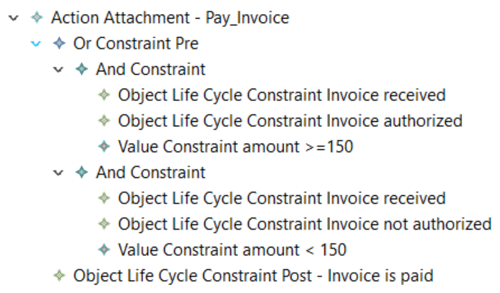

Action Attachment - Pay_Invoice
Or Constraint Pre
And Constraint
Object Life Cycle Constraint Invoice received
Object Life Cycle Constraint Invoice authorized
Value Constraint amount >=150
And Constraint
Object Life Cycle Constraint Invoice received
Object Life Cycle Constraint Invoice not authorized
Value Constraint amount < 150
Object Life Cycle Constraint Post - Invoice is paid

Figure 4.9: Complex semantic constraint

4.6 Annotation of Process Models

For annotation of process models we use the UML Profile given in Figure 4.10. This profile allows attaching semantic constraints to *Actions* and *Control Flows* as well as an associated OLC behavior to *Actions*.

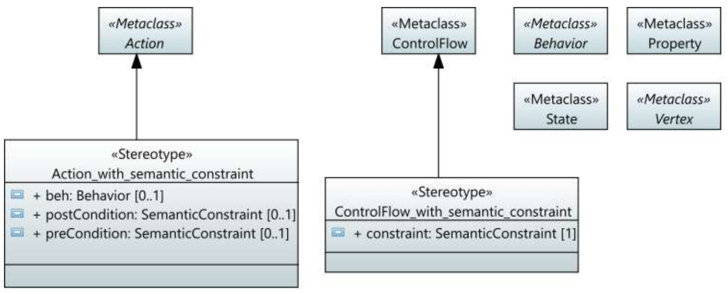

«Metaclass» Action
«Metaclass» ControlFlow
«Metaclass» Behavior
«Metaclass» Property
«Metaclass» State
«Metaclass» Vertex
«Stereotype» Action_with_semantic_constraint
+ beh: Behavior [0..1]
+ postCondition: SemanticConstraint [0..1]
+ preCondition: SemanticConstraint [0..1]
«Stereotype» ControlFlow_with_semantic_constraint
+ constraint: SemanticConstraint [1]

Figure 4.10: Profile used for annotation of the process models (Activity diagrams) – for specification of *Semantic Constraint* see Figure 4.7

The reason for using a profile and stereotypes instead of directly modifying the Activity diagram is as follows. The premise of our workflow is that the process models are not exclusively used for verification. Instead, they are also used elsewhere in the development process, e.g., for generating the implementation code. Therefore, there may be some pre- and postconditions attached already. We consider those conditions depended on the target platform and, therefore, not directly useable for connecting process models with OLCs. At the latest when it comes to behavior an approach without profile runs into problems, since all types of *Actions* have slightly different capabilities when it comes to referencing or defining behavior. There would have to be a separate solution for each type of action. We also deem a *UML Profile* defining *Stereotypes* a much clearer solution in terms of (human) validation of annotated process models.

The annotation step is implemented via model-to-model transformation implemented in the Operational Mappings Language defined as part of MOF2 Query/View/Transformation (QVT) [23]. In this step, the given process models are taken and *Actions* and *Control Flows* referenced in the *Semantic Specification* are stereotyped.

Figure 4.11 shows the relation between the relevant input files of the workflow and the generated *Annotated Process Models*. As already stated above, the *Semantic Specification* is the key part for connecting process models and OLCs by referencing elements defined by them.

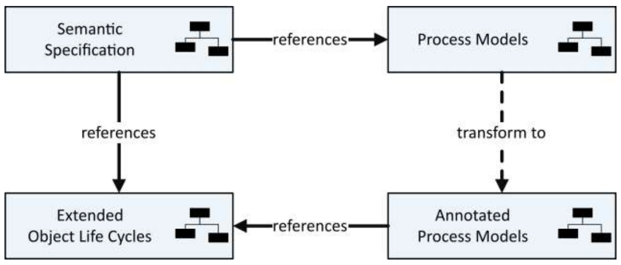


Figure 4.11: Relation between process models, OLCs and semantic specification

The actual transformation takes the semantic specification and the process models as inputs and generates a new file containing the annotated process models. The (original) process model keeps independent from OLCs, which means independent from a specific context the process runs in. Only the annotated process models depend on the OLCs referenced by the semantic specification. This allows, besides the separation of concerns, reusing process models and OLCs in different combinations.

4.7 Model to nuXmv-Code Generation

In this section we explore how annotated process models and OLCs are transformed into nuXmv code to allow model checking. It starts with a general overview of the concept we use to define nuXmv models. After that, we show how an entire Activity diagram is built from its individual components, like actions and control flows, using this concept. After that, we present how those individual components are realized in nuXmv code.

4.7.1 General Concept

For structuring the generated nuXmv source code already in the course of the transformation, we make use of *Modules*, which are custom, pre-defined source code constructs that can be used with nuXmv. The nuXmv manual [27] defines a *Module* declaration as an encapsulated collection of declarations, constraints and specifications, which is comparable to a class in object-oriented programming languages. A Module can be instantiated as many times as necessary, each of which creates a new identifier scope. During instantiation, parameters may be passed to Modules for setting values. Modules can contain instances of other Modules as well, which enables the construction of hierarchical structures [47].

Another concept we heavily relay on is the connection of instances of Modules via variables declared outside of the scope of those Modules. This allows us to define generic Modules, in the sense of independent from the actual variables used for connecting.

For better understanding of the code snippet later given in this thesis, we first look at the simple nuXmv example involving two counters. The first counter has a value range from 0 to 3 and is enabled permanently, therefore increases its value in every step. The second counter has a value rang from 0 to 15 and increases its value only if the first counter has its maximum value of 3.

Listing 4.3: Example with two Counters – Counter Module

```
MODULE Counter(maxVal, overflowSig, enable, reset)
  VAR
    state : 0..maxVal; -- define variable holding value of the counter
  ASSIGN
    init(state) := 0; -- initialize the counter with value 0
    next(state) :=
      case
        reset : 0; -- reset counter if external event occurs
        state = maxVal & enable : 0; -- reset counter if maximal value is reached
        state < maxVal & enable : state + 1; -- increase counter value
        TRUE : state; -- mandatory default case
      esac;

    overflowSig := state = maxVal; -- signal overflow of the counter
```

For implementing those counters, we define the nuXmv Module *Counter*, given in Listing 4.3, that is parameterizable via the parameters *maxVal*, *overflowSig*, *enable* and *reset*. *maxVal* is the maximum value the counter can reach. When this value is reached it is reset to 0 in the next step. *overflowSig* is a “reference” to a variable of type Boolean, which is used to signal an overflow. The constants or variables of type Boolean *enable* and *reset* enable counting or reset the counter, respectively.

With this definition, we are now able to declare the two counters in the *main* Module. The *main* Module is a mandatory Module, without any parameters, that serves as the starting point of each nuXmv model and sets up the entire model.

Listing 4.4: Example with two Counters – Main Module

```
MODULE main
  DEFINE
    maxVal := 3; -- define symbol/constant maxValue
  VAR
    sigC1 : boolean; -- variable signaling overflow counter 1
    sigC2 : boolean; -- variable signaling overflow counter 2
    c1 : Counter(maxVal, sigC1, TRUE, FALSE); -- counter 1
    c2 : Counter(15, sigC2, sigC1, FALSE); -- counter 2

SPEC AG(c2.state < 4);
```

In our case the *main* Module, given in Listing 4.4, first defines the identifier *maxVal* for the expression 3. After that the VAR section is used to declare to Boolean variables (*sigC1* and *sigC2*) as well as to instances of the *Counter* module named *c1* and *c2*. Each of these instances is parametrized accordingly. For example the second counter is declared by *c2 : Counter(15, sigC2, sigC1, FALSE);*. Setting an maximum value of 15, using *sigC2* as *overflowSig* of the counter, *sigC1* to enable the counter and assigns the permanent value FALSE to *reset*.

To actually connect the two counters the variable *sigC1* is used as highlighted in Listing 4.5. With this *sigC1* is assigned in counter *c1* and is (later) used in counter *c2* to decide if the counter value should be increased or not.

Listing 4.5: Connecting the two counters

```
c1 : Counter(..., sigC1, ..., ...);
c2 : Counter(..., ..., sigC1, ...);
```

This concept, of assigning a value to a variable in one instance of a Module and use the same variable to enable/trigger some behavior in another instance, is used e.g. for connecting actions via control flows.

This example also shows another concept we highly relay on, synchronization of initially asynchronous FSMs via variables/signals. Counters *c1* and *c2* are synchronized via the signal *sigC1*, to increase the value of counter *c2* only if the counter *c1* is going to overflow in the next step. Given that the state values are combined properly this, in essence, creates an combined counter with a value range from 0 to 63.

4.7.2 Activity Diagram to Code

As described above, an *Activity* describes a token system. In this section, it is explained how that token system is realized by a module that is composed of instances of other modules. However, in contrast to the pre-defined modules that represent the building blocks of the *Activity*, e.g., *Control Flow* and *Actions*, modules representing activities must be generated on a case-by-case basis. Thus, this section can show only the general concept for building such modules here.

Let's look at the module definition given in Listing 4.6. It implements the *Activity* modeling the *Delivering Company Process*.

Listing 4.6: nuXmv Module representing the *Delivering Company Process*

```
MODULE Delivering_Company_Process(controlFlowIn, controlFlowOut, pre, post,
    container, resetExt)
 VAR
  -- Instantiate Activity Edges
  InitialNodetoCreate_Invoice_15: ControlFlow();
  Book_InvoicetoFinalNode_16: ControlFlow();
  Create_InvoicetoTransmit_Invoice_17: ControlFlow();
  Transmit_InvoicetoReceive_Paid_Invoice_18: ControlFlow();
  Receive_Paid_InvoicetoBook_Invoice_19: ControlFlow();

  -- Instantiate Activity Nodes
  InitialNode: InitialNode(controlFlowIn.forwardSignal & pre,
      InitialNodetoCreate_Invoice_15, reset);
  Create_Invoice: Action(container.olc.DCPInvoice_Errorobj.Created =
      Created_False, container.olc.DCPInvoice_Errorobj.Created = Created_True,
      InitialNodetoCreate_Invoice_15, Create_InvoicetoTransmit_Invoice_17, reset)
      ;
  Book_Invoice: Action(TRUE, TRUE, Receive_Paid_InvoicetoBook_Invoice_19,
      Book_InvoicetoFinalNode_16, reset);
  FinalNode: ActivityFinalNode(controlFlowOut.backwardSignal,
      Book_InvoicetoFinalNode_16, reset);
  Transmit_Invoice: SendAction(TRUE, TRUE, Create_InvoicetoTransmit_Invoice_17,
      Transmit_InvoicetoReceive_Paid_Invoice_18, reset);
  Receive_Paid_Invoice: ReceiveAction(TRUE, TRUE, Receive_Paid_InvoiceTrig,
      Transmit_InvoicetoReceive_Paid_Invoice_18,
      Receive_Paid_InvoicetoBook_Invoice_19, reset);

 ASSIGN
  controlFlowIn.backwardSignal := controlFlowIn.forwardSignal & InitialNode.
      ACTIVE;
  controlFlowOut.forwardSignal := FinalNode.FINISHED & post;

 DEFINE
  trig6 := Create_Invoice.BEH;
  trig7 := Book_Invoice.BEH;
  trig8 := Transmit_Invoice.TRIGGEROUT;
  trig9 := Receive_Paid_Invoice.BEH;
  trig10 := Transmit_Invoice.TRIGGEROUT;
  Receive_Paid_InvoiceTrig := container.trig14;
  SUB_ACTIVE := InitialNode.ACTIVE | Create_Invoice.ACTIVE | Book_Invoice.ACTIVE
      | FinalNode.ACTIVE | Transmit_Invoice.ACTIVE | Receive_Paid_Invoice.ACTIVE;
  reset := resetExt | controlFlowOut.backwardSignal;
```

The module of an *Activity* has 6 parameters:

- controlFlowIn ... *Control Flow* connecting the previous action (or *Activity Node*) to this action.
- controlFlowOut ... *Control Flow* connecting this action to the subsequent *Activity Node*.
- pre ... Precondition as given by the semantic specification. If no precondition is given, TRUE has to be given.
- post ... Postcondition as given by the semantic specification. If no postcondition is given, TRUE has to be given.
- container ... Instance of a module instantiating it or an instance even further up the chain.
- resetExt ... Signal that resets all actions in the activity and the activity itself.

The body of the module consists of three main sections. In the first section, all instances of the individual components – *Activity Edges* and *Activity Nodes* – the activity is composed of are created. In order to achieve the desired behavior as defined in the corresponding UML Activity diagram, the parameters of each instance have to be set to the right value. However, this turned out to be quite easy since it mainly means determining which *Control Flows* are connected to an *Activity Node* and setting the pre- and postcondition of an action according to its semantic specification. The other two sections contain additional code dealing with handling the incoming and outgoing control flow of the activity – ASSIGN section – as well as defining signals for triggering behavior of object life cycles, error handling and flags that enable an easier analysis of counterexamples – DEFINE section.

An advantage of using instances of modules to assemble an activity is that the implementation is much easier to read and understand. Especially when directly compared to the corresponding Activity diagram. Looking at the Activity diagram modeling the *Delivering Company Process* of our running example, given in Figure 2.2, and comparing it to the generated implementation, given in Listing 4.6, shows that each element in the diagram directly corresponds to a single line of code.

After defining such a Module for an activity it then can be used to define even more complex behavior, instantiating it whenever a *Call Behavior Action* is used in another activity, like the *Main Activity* given in Figure 4.5. Listing 4.7 shows a stripped-down version of the implementation of it. As one can see, on the level of the nuXmv Module, including an entire activity into another one is not different from using other types of *Activity Nodes*.

Listing 4.7: Stripped-down version of the nuXmv Module representing the *MainActivity*

```
MODULE MainActivity(controlFlowIn, controlFlowOut, pre, post, container, resetExt
    )
  VAR
    -- Instantiate Activity Edges
    ...

    -- Instantiate Activity Nodes
    ...
    CallDeliveringCompanyProcess: Delivering_Company_Process(
        ForkNodetoCallDeliveringCompanyProcess_1,
        CallDeliveringCompanyProcesstoJoinNode_3, TRUE, TRUE, container, reset);
    CallCustomerCompanyProcess: Customer_Company_Process(
        ForkNodetoCallCustomerCompanyProcess_2,
        CallCustomerCompanyProcesstoJoinNode_5, TRUE, TRUE, container, reset);

  ASSIGN
    ...

  DEFINE
    ...
```

4.7.3 Control Flow to Code

To better understand all other mapping from Activity diagrams elements to nuXmv code we first take a look how *Control Flows* are mapped to nuXmv code.

The underlying concept when it comes to control flow in *Activities* are tokens[6, Sec. 15.2.3.1]. Although tokens are not explicitly modeled in UML, it is said that a *Control Flow* passes tokens from its *source* to its *target* – both of type *Activity Node*. Thereby, a *Control Flow* never actually holds a token by itself.

Instead tokens offered to a *Control Flow* by the source *Activity Node* may not immediately flow along the edge. Tokens only move when the offer is accepted by the *Control Flow*, which requires at least the target *Activity Node* to accept them. [6, Sec. 15.2.3.1]

To emulate this behavior as good as possible the nuXmv Module representing the *Control Flow* given in Listing 4.8 defines two signals – *forwardSignal* offers a token to the target of the *Control Flow* and *backwardSignal* is used to signal back that the offer is accepted.

Listing 4.8: nuXmv Module representing a *Control Flow*

```
MODULE ControlFlow
    VAR
        forwardSignal : boolean;
        backwardSignal : boolean;
```

Proper coordination of *forwardSignal* and *backwardSignal* guaranties that the token is handled appropriately – e.g. not duplicated. However, this is the sole responsibility of the *Activity Nodes*.

4.7.4 Actions to Code

The models of Activity diagrams as sketched above are transformed automatically to several synchronized FSMs, more precisely asynchronous FSMs synchronized by signals.

Each Action in an Activity diagram is transformed to a corresponding FSM part. There are four states created for each Action, as illustrated in Figure 4.12, and transitions between them.

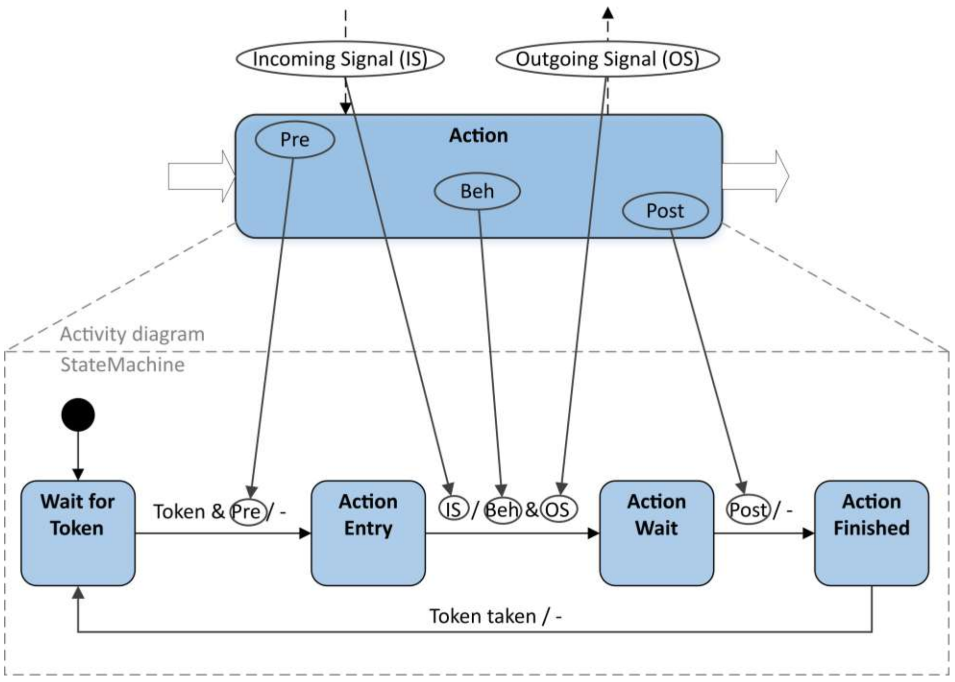


Figure 4.12: Transformation of an Action to FSM (Extended version of the transformation used in [47])

The first state *Wait for Token* is the entry state of the action. This state is necessary since UML does not allow guard and triggers to be attached to control flows outgoing of *initial Pseudostate* and represents the state of the action in which it waits until it is activated. The next state *Action Entry* becomes necessary to represent waiting for an *Incomming Signal*. This is needed for *Receive Event Actions* since technicality those actions are active if the hold a control flow token but have to wait anyway until the (asynchronous) signal was received. *Action Wait* denotes that state of waiting until the postcondition is fulfilled. Last but not least in state *Action finished* the action has finished its operation and waits till the control flow token is taken by the following action (or *Activity Node*). In all state with the exception of *Wait for Token* the action holds the control flow token and is therefore considered active in the sense that it is ether actively waiting for e.g an *Incoming Signal* or performs some behavior.

The transitions between those states define the actual behavior of the action. Starting in the action initial state *Wait for Token* – action is inactive – the action is only activated if the precondition is met and, at the same time, the preceding action (or *Activity Node*) offers a control flow token to it. This is modeled via the guard condition, *Token & Pre* on the incoming transition of *Action Entry*. The outgoing transition of *Action Entry* (which is equal to the incoming transition of *Action Wait*) does the heavy lifting by triggering the associated behavior of an OLC (Beh) as well as sending the *Outgoing Signal* (OS). However, this transition is only taken if the *Incoming Signal* (IS) is received or was previously received. Since we explicitly want to check whether an action has the desired effect the transition, leading to *Action Finished*, checks the postcondition (Post) defined in the semantic specification. This guaranties that the intended outcome of the behavior has been achieved. The last transition from *Action Finished* back to *Wait for Token* is taken when the control flow token is taken by the next action (or *Activity Node*).

As you may have noticed for the reason of simplification of Figure 4.12 self-loops have been omitted. It is implicitly assumed that the FSM stays in its current state when none of the given transitions are taken. Later we will see that this is not entirely true, since the corresponding nuXmv Module also has a way to be reset, which is also not shown in this representation.

The module definition given in Listing 4.9 shows how this state machine is implemented as nuXmv code.

Listing 4.9: nuXmv Module representing an *Action*

```
MODULE Action(Pre, Post, triggerIn, controlFlowIn, controlFlowOut, TIMEOUT)
  VAR
    state : {initial, S1, S2, S3};
    triggerBuffer : boolean;
  ASSIGN
    init(state) := initial;
    next(state) :=
      case
        TIMEOUT : initial;
        state = initial & controlFlowIn.forwardSignal & Pre: S1;
        state = S1 & (triggerIn | triggerBuffer) : S2;
        state = S2 & Post : S3;
        state = S3 & controlFlowOut.backwardSignal : initial;
        TRUE : state;
      esac;

    init(triggerBuffer) := FALSE;
    next(triggerBuffer) :=
      case
        state = S1 & next(state) = S2 : FALSE;
        triggerIn : TRUE;
        TRUE: triggerBuffer;
      esac;

    controlFlowOut.forwardSignal := (state = S3) & !TIMEOUT;
    controlFlowIn.backwardSignal := (state = initial) & controlFlowIn.
        forwardSignal & Pre & !TIMEOUT;

  DEFINE
    ACTIVE := (state != initial);
    BEH := (state = S1) & (triggerIn | triggerBuffer) & !TIMEOUT;
    TRIGGEROUT := (state = S1) & !TIMEOUT;
```

The module *Action* has 6 parameters:

- pre ... Precondition as given by the semantic specification. If no precondition is given, TRUE has to be given.
- post ... Postcondition as given by the semantic specification. If no postcondition is given, TRUE has to be given.
- triggerIn ... *Incoming Signal* of the *Receive Actions*.
- controlFlowIn ... *Control Flow* connecting the previous action (or *Activity Node*) to this action.
- controlFlowOut ... *Control Flow* connecting this action to the subsequent *Activity Node*.
- TIMEOUT ... Signal that resets the action to its initial state.

In addition to the parameters we consider the following definitions as as part of the interface of the module:

- ACTIVE ... Signals that the action is active. (Currently holds a token.)
- BEH ... Used to trigger the behavior given in by the semantic specification.
- TRIGGEROUT ... Sends asynchronous signal.

To keep the individual lines of code shorter the states of the FSM named differently from the states in Figure 4.12. To be more precise state *Wait for Token*, *Action Entry*, *Action Wait* and *Action Finished* were renamed to *initial*, *S1*, *S2* and *S3*, respectively. In addition to that the code includes also some sort of buffer to keep track of whether an *Incoming Signal* has been send. This is necessary since the nature of those signals is asynchronous and thus may be received then the action is in another state then *Action Entry/S1*. The buffer called *triggerBuffer* listens independently from the state of the action whether a signal was received. If the signal is not directly used to trigger the transition from *Action Entry/S1* to *Action Wait/S2*, the buffer stores the information that an *Incoming Signal* was received but not used. Later, when the action reaches *Action Entry/S1* the buffer triggers the transition and resets itself.

We later found that the combination of *Send Signal Actions*, *Receive Event Actions* and *Actions* in one nuXmv Module is not reasonable since for every type, parts of the functionality are not used. Those parts would increase the complexity of the model due to unnecessarily increasing the number of variables. Therefore, we used three different modules – one Module for each type of action[1].

[1]Particularly interested readers are revered to Listings B.2, B.3 and B.4 of the appendix.

4.7.5 Control Nodes to Code

In contrast to *Executable Nodes*, such as all types of actions, tokens cannot "rest" at *Control Nodes*. Instead tokens are just routed between incoming and outgoing edges. As we will later see there are two exceptions from that rule, *Initial Nodes* and *Fork Nodes*.

The modules defined to implement the different types of *Control Node* try to be as accurate as possible. However, some types have a broad definition including multiple scenarios differing by the number and type of incoming and/or outgoing *Activity Edges*. E.g. a *Decision Node* has at least one incoming edge but allows up to two, with different meanings[6, Sec. 15.3.3.6]. During the remainder of this thesis we only consider the most basic implementation using only *Control Flows*.

Initial Node Since the *Initial Node* is the starting point of an activity the node's implementation must generate a token. However, an activity may not be triggered immediately after the initialization of the software/model. Therefore, the nuXmv implementation includes a signal to enable the token generation. This approach also enables us to trigger the activation of an activity as needed. To avoid generating multiple tokens even if only one token generation is intended, a "safeguard" was included. The *token-generator* introduced in Listing 4.10 does this by generating tokens only if the *createTokenSignal* changes from FALSE to TRUE. As long as the *Initial Node* holds a token, it is also offered to the outgoing flow. The node is deactivated immediately after the offer is accepted.

Listing 4.10: nuXmv Module representing an *Initial Node*

```
MODULE InitialNode(createTokenSignal,controlFlowOut, TIMEOUT)
  VAR
    state_tokengenerator : {S1, S2, S3};
    state : {active, inactive};

  ASSIGN
    init(state_tokengenerator) := S1;
    next(state_tokengenerator) :=
      case
        state_tokengenerator = S1 & TIMEOUT : S1;
        state_tokengenerator = S2 & TIMEOUT : S3;
        state_tokengenerator = S1 & createTokenSignal : S2;
        !createTokenSignal : S1;
        TRUE : state_tokengenerator;
      esac;

    init(state) := inactive;
    next(state) :=
      case
        TIMEOUT : inactive;
        state_tokengenerator = S1 & createTokenSignal : active;
        state = active & controlFlowOut.backwardSignal : inactive;
        TRUE : state;
      esac;

    controlFlowOut.forwardSignal := (state = active) & !TIMEOUT;

  DEFINE
    ACTIVE := (state = active);
```

Activity Final Node The *Activity Final Node* consumes a token offered on the incoming flow and tunnels it up to the higher-level activity called the behavior. We will later see how exactly this works in the context of the whole activity. However, a "safeguard" like the one in the *Initial Node* module was introduced to prevent missing a token.

Listing 4.11: nuXmv Module representing an *Activity Final Node*

```
MODULE ActivityFinalNode(consumeTokenSignal, controlFlowIn, TIMEOUT)
  VAR
    state_consumer : {S1, S2, S3};
    state : {active, inactive};

  ASSIGN
    init(state_consumer) := S1;
    next(state_consumer) :=
      case
        state_consumer = S1 & TIMEOUT : S1;
        state_consumer = S2 & TIMEOUT : S3;
        state_consumer = S1 & consumeTokenSignal : S2;
        !consumeTokenSignal : S1;
        TRUE : state_consumer;
      esac;

    init(state) := inactive;
    next(state) :=
      case
        TIMEOUT : inactive;
        state = inactive & controlFlowIn.forwardSignal : active;
        state_consumer = S1 & consumeTokenSignal : inactive;
        TRUE : state;
      esac;

    controlFlowIn.backwardSignal := state = inactive & controlFlowIn.
        forwardSignal
        & !TIMEOUT;

  DEFINE
    FINISHED := (state = active);
    ACTIVE := (state = active);
```

Fork Node The *Fork Node* implementation given in Listing 4.12 splits an incoming token into two. The output tokens are provided without delay as soon as a token is offered on the incoming flow. However, the *Fork Node* does not accept the token offered by the incoming flow as long as at least one outgoing flow accepts the offer. If both outgoing flows accept the offer simultaneously, everything is fine and the *Fork Node* never actually holds a token. However, if only one offer is accepted, the *Fork Node* holds the remaining token. For this purpose, the variables *token1* and *token2* are used.

Listing 4.12: nuXmv Module representing a *Fork Node*

```
MODULE Fork(controlFlowIn, controlFlowOut1, controlFlowOut2, TIMEOUT)
  VAR
    token1 : boolean;
    token2 : boolean;

  ASSIGN
    init(token1) := FALSE;
    next(token1) :=
      case
          TIMEOUT : FALSE;
          controlFlowOut1.backwardSignal : FALSE;
          controlFlowIn.backwardSignal : TRUE;
          TRUE : token1;
      esac;

    init(token2) := FALSE;
    next(token2) :=
      case
          TIMEOUT : FALSE;
          controlFlowOut2.backwardSignal : FALSE;
          controlFlowIn.backwardSignal : TRUE;
          TRUE : token2;
      esac;

    controlFlowOut1.forwardSignal := (controlFlowIn.forwardSignal & !block_direct
        ) | (token1 & !TIMEOUT);
    controlFlowOut2.forwardSignal := (controlFlowIn.forwardSignal & !block_direct
        ) | (token2 & !TIMEOUT);
    controlFlowIn.backwardSignal := controlFlowIn.forwardSignal & !block_direct
        & (controlFlowOut1.backwardSignal | controlFlowOut2.backwardSignal);


  DEFINE
    block_direct := (token1 = TRUE | token2 = TRUE | TIMEOUT);
    ACTIVE := ((token1 = TRUE) | (token2 = TRUE));
    DIRECT := controlFlowIn.forwardSignal & !block_direct & !TIMEOUT;
```

Join Node The implementation is done in such a way that on the outgoing flow, a token is offered only if the *Join Node* is offered a token on both incoming flows. Both offers are accepted simultaneously if the offer on the outgoing flow is accepted by the next *Activity Node*. The *Join Node* never actually holds a token itself.

Listing 4.13: nuXmv Module representing a *Join Node*

```
MODULE Join(controlFlowIn1, controlFlowIn2, controlFlowOut, TIMEOUT)
  ASSIGN
    controlFlowOut.forwardSignal := ACTIVE & !TIMEOUT;
    controlFlowIn1.backwardSignal := controlFlowOut.backwardSignal & !TIMEOUT;
    controlFlowIn2.backwardSignal := controlFlowOut.backwardSignal & !TIMEOUT;

  DEFINE
    ACTIVE := controlFlowIn1.forwardSignal & controlFlowIn2.forwardSignal
        & !TIMEOUT;
```

Merge Node A *Merge Node* has to offer a token to the output flow for each incoming token. Therefore, the implementation given in Listing 4.14 accepts one incoming offer, even as the *Merge Node* is offered a token on both incoming flows. This avoids losing tokens. However, for our implementation, we decided to prioritize a token offered on *contolFlowIn1* over a token offered on *contolFlowIn2*.

Listing 4.14: nuXmv Module representing a *Merge Node*

```
MODULE Merge(controlFlowIn1, controlFlowIn2, controlFlowOut, TIMEOUT)
  ASSIGN
    controlFlowOut.forwardSignal := ACTIVE;
    controlFlowIn1.backwardSignal := controlFlowIn1.forwardSignal &
        controlFlowOut.backwardSignal & !TIMEOUT;
    controlFlowIn2.backwardSignal := controlFlowIn2.forwardSignal &
        !controlFlowIn1.forwardSignal & controlFlowOut.backwardSignal & !TIMEOUT;

  DEFINE
    ACTIVE := (controlFlowIn1.forwardSignal | controlFlowIn2.forwardSignal) &
        !TIMEOUT;;
```

Decision Node For implementing the *Decision Node*, we decided to pull the guards defined on the outgoing *Control Flows* into the decision itself. The main reason was to guarantee that, at most, one outgoing flow is offered a token. The conditions themselves are given via the parameters *Cond1* and *Cond2* of the module. The variable *decision* is used to control where the token is forwarded to. It is necessary for the case if both conditions are met simultaneously. For this case, the module is set up so that the model-checker checks both non-deterministic outcomes – token offered to the *controlFlowOut1* or *controlFlowOut2*.

Listing 4.15: nuXmv Module representing a *Decision Node*

```
MODULE Decision(Cond1, Cond2, controlFlowIn, controlFlowOut1,
    controlFlowOut2, TIMEOUT)
  VAR
    decision : {block, out1, out2};

  ASSIGN
    decision :=
      case
        Cond1 & Cond2 : {out1, out2};
        Cond1 : out1;
        Cond2 : out2;
        TRUE : block;
      esac;
    controlFlowOut1.forwardSignal := controlFlowIn.forwardSignal &
        decision = out1 & !TIMEOUT;
    controlFlowOut2.forwardSignal := controlFlowIn.forwardSignal &
        decision = out2 & !TIMEOUT;
    controlFlowIn.backwardSignal := controlFlowIn.forwardSignal &
        (controlFlowOut1.backwardSignal | controlFlowOut2.backwardSignal) &
        !TIMEOUT;

  DEFINE
    ACTIVE := controlFlowIn.forwardSignal;
```

As you may have noticed all implementations – *Control Nodes* as well as *Actions* – strongly depend on the assumption that a *backwardSignal* on a *Control Flow* immediately clears the *forwardSignal*. Is this assumption violated an unintentional replication of tokens would occur during the transition of the token form one node to another.

4.8 Properties for Checking-Consistency

With the approach given in this thesis, the consistency of process models with connected object life cycles can be verified. For this consistency verification, predefined properties are used. However, a general property related to error states is necessary for consistency verification, which is predefined in our approach. Essentially, we can verify if the process defined through actions is consistent with a given object life cycle, where the process defines the usage of the actions. We define *consistency* using the following criteria [47]:

1. *Can the process model (or a defined part of it) continue with an action based on the given state of the attribute values in the extended object life cycle?* Answering this question necessitates the semantic specifications as well as the order in which actions are executed. An action can only be executed if its precondition is met. If the *precondition* is not met, i.e., an attribute of the object life cycle has a different state, the process model gets stuck and cannot continue. Our approach to model-check this actually checks whether the process can reach on all paths one of certain defined states $DefStates$, i.e., either a final state of the whole process or a defined end state of a part of the process to be checked for consistency (rather than the completeness of the whole process). If so, it did not get stuck before. The nuXmv tool can check this through a generic property $F(DefStates)$.

   Even if the precondition is fulfilled, the process model can only continue if also its *postcondition* is fulfilled, i.e., an attribute of the object life cycle is set. Since the state machines defining the behavior of actions are constructed in such a way that it waits till the *postcondition* is met, the fulfillment of it can be checked using the same property as given above.

2. *Can the object life cycle in a given state handle a given action in the process model?* This criterion ensures that no action triggers a behavior of an object life cycle that it cannot handle in its current state, e.g., transmitting the invoice before it is even created. Our approach for detecting such cases is to transition the object life cycle into an error state and never leave it again. However, defining such error states is optional since not all attempts to trigger a certain behavior of a OLC in a “wrong” state may be considered incorrect. Overall, if such an error state is defined, this criterion is checked through a generic property $G(\neg Error)$, which essentially states that never an error state is reached.

The actual consistency verification with the nuXmv tool is based on these criteria and uses the related properties, i.e., these properties are used as input in combination with the generated nuXmv model for the “Verification” step in Figure 4.4. Only if they are all fulfilled, then the models are consistent with each other. This complete verification run (for checking our two consistency properties) takes a few seconds on a usual laptop computer. The one we used for all our model-checking runs reported in this chapter of this thesis has an Intel(R) Core(TM) i7-6920HQ CPU @ 2.90GHz with 16 GB of RAM, running Windows 10 64-bit [47].

For example, changing the precondition of the *Receive Paid Invoice* action to additionally contain *booked(Invoice)* leads to inconsistency. With this additional precondition, the FSM of the delivering company cannot continue past the *Receive Paid Invoice* action. Therefore, it cannot reach the state *Receive Paid Invoice Entry* since the condition on the incoming transition is not fulfilled (criterion 1). Thus, the remainder of the FSM cannot be reached, which is checked by the property $F(DefStates)$. Criterion 1 is also not fulfilled if the process model itself is inconsistent, even if all the pre- and postconditions would fit. For example, the model-checking result is analogous to the one above if the action *Receive Paid Invoice* is deleted from the process model, since the precondition *booked(Invoice)* of the action *Book Invoice* will not become true.

As an example related to criterion 2, if we change the signal of the action *Book Invoice* to the state *Set Transmitted* instead of *Set Booked*, then there is no applicable transition for this signal in the previous state *Set Payment Received* of the object life cycle. In this case, the object life cycle transitions into the *error* state *Error DP*. This violates criterion 2, which is checked by the predefined error state property $G(\neg Error)$. Analogously, criterion 2 is not fulfilled, if an inconsistent transition condition is specified in the object life cycle [47].

4.9 Case Study and Results

For investigating the practical applicability of our approach, we sketch a case study of a real-world cyber-physical process for charging an electric vehicle at a charging station[2]. This case study involved several iterations of executing the workflow defined in Section 4.2 and improving the process based on the result of the analysis until the process model was consistent with the object life cycles (OLCs) and considered valid by a domain expert.

First, a process model was created, including OLCs and semantic action specifications. It was checked against the consistency properties defined above. After a few iterations of resolving consistency violations, the resulting version of the process model was successfully verified for consistency. Then we consulted a domain expert for external validation, i.e., whether this process is right for practice. The feedback from the domain expert led to another version, and after a few more iterations, it was successfully verified for consistency. The domain expert had only one remark on this version, which was addressed by another small change. The resulting version was successfully verified for consistency and finally validated by the domain expert. Figure 4.13 shows this version of the process model for charging an electric vehicle.

[2]Most of this section is directly taken from [47, Sec. 9].

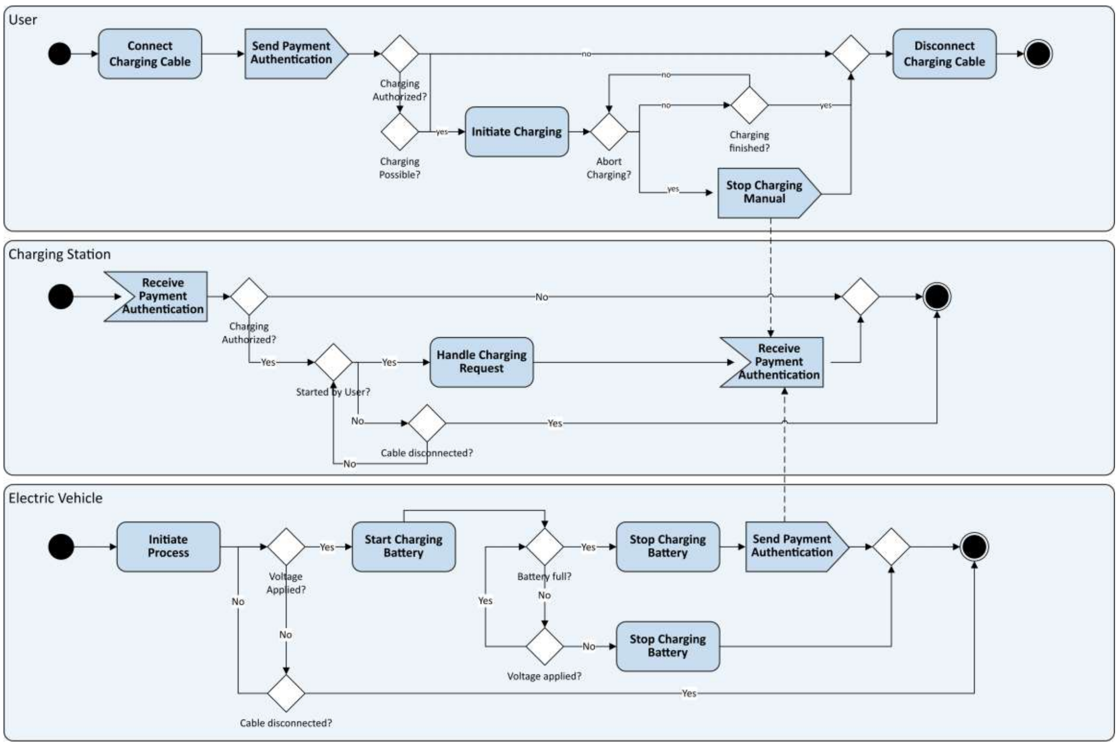


Figure 4.13: Activity diagram of a process for charging an electric vehicle at a charging station - drawn after [47, Fig. 9]

This electric charging process consists of three sub-processes for the actors *User*, *Charging Station* and *Electric Vehicle*, respectively. These sub-processes are executed concurrently and, like the process of our running example, they are synchronized via send- and receive actions at specific points. For example, *Charging Station* waits for authentication of the payment before actually initiating the charging.

In addition, these sub-processes communicate based on physical interaction modeled through extended life cycles. For example, connecting the charging cable in the user process triggers the other sub-processes based on the communication between the extended life cycle objects.

These are all *extended* object life cycles according to our approach. We just show the FSM of an attribute of the *Charging Station* object life cycle as an example, for illustrating an important difference to the attribute FSMs of the running example above. In Figure 4.14, it is easy to see that the attribute for representing the state of connection of the charging cable can be *set* (cable connected) *and reset* (cable not connected). This example shows that the model in the case study is a *non-monotonic system* (in contrast to the monotonic system of the running example above, which was only able to remember having reached a certain state by use of the corresponding attribute).

Another important difference of this process model to the one of the running example is its inclusion of simple *loops*. They model reacting to user interaction and external

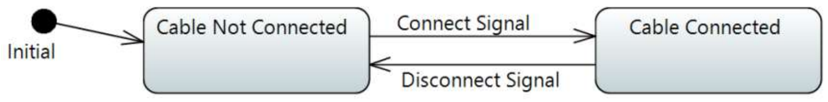


Figure 4.14: Attribute of an (extended) object life cycle that can be set and reset

events, e.g., with regard to whether the cable is being disconnected by the user. Note, that using non-monotonic attributes is necessary for the termination of such loops. As an example, let us consider the *User* process and the attribute representing the state of connection of the charging cable. First, the user connects the charging cable between the charging station and the electric vehicle. Listing 4.16 shows the semantic action specification of *Connect Charging Cable* semi-formally.

Listing 4.16: Extended Semantic Action Specification of Connect Charging Cable Action (extended version of Listing 8 in [47] which specifies the behavior of the action explicitly.)

```
Connect Charging Cable:
   Pre: disconnected(cable)
   Post: connected(cable)
   Beh: call(cable, ConnectCable)
```

After charging has finished, the user disconnects the cable, which is reflected in the model by resetting the corresponding attribute to “Cable Not Connected”. In addition to this normal operation, however, the user may at any time during the charging process disconnect the cable, also before the charging process has finished. This is taken care of in the process model by the loop between decisions “Abort Charging?” and “Charging Finished?” in Figure 4.13. This loop is eventually terminated by either resetting to “Cable Not Connected” (cf. non-monotonic attribute) or if charging has finished.

Let us focus on the verification done in the course of this case study. For verifying the consistency of this charging process, we had to generate the input source code from the models given as UML Activity diagrams and UML State Machines. This was done using our implementation of the automatic transformation as described above. The actual verification was done using bounded model checking in nuXmv 2.0.0, where we verified both consistency properties.

Especially in the first iterations of developing this process model, model checking revealed several consistency problems. For example, an early version of the *Electric Vehicle* sub-process model was specified inconsistently, as a key action, *Initiate Process*, was missing. This prevented the process to end in its designated state. Other errors included a postcondition triggering a wrong transition (cable disconnect instead of connect) in the *Charging Station* object life cycle, essentially halting the process. In another example, the object life cycle was inconsistent as the attributes and their transitions were modeled wrongly, i.e., a wrong signal on the disconnect cable transition.

Let us elaborate on yet another example, where an error was made in the course of copy&edit, when a precondition was erroneously not deleted. This additional precondition ($Battery\_Level = Full$) for checking the battery level in the Initiate Process Action of the Car process prevented this process from continuing if the battery was not already charged. Hence, the battery could never be charged and remained in its initial state, much as the Car process itself. This, in turn, led to an infinite loop as the decision elements in the other processes never evaluated to true.

The nuXmv tool recognized this situation and reported a counterexample. Such a counterexample is a list of states showing the trace that leads to an error/property violation. Each listed state is a snapshot of value assignments, e.g., state of parameters, in the course of model checking. That is, each state differs from the next one by a state transition of an FSM or an attribute change. For example, in our models an $ACTIVE$ variable indicates for each Action whether it is currently being executed or not. Hence, a counterexample for our models shows a trace of Actions having been activated.

We found that analyzing such changes of activations, or their absence can facilitate understanding the problem that caused a counterexample, e.g., if a process got stuck. In the case of this particular example, we first analyzed all ACTIVE states and realized that the active state of the Car Process has not been changed.

Analyzing such counterexample traces manually can require much effort, since they are often verbose. With an increasing number of variables (or in our case Actions), more and more entries are listed in the counterexample trace. For easing the task of analyzing these traces, we developed a simple filtering tool (in Python), which allows focusing on certain text strings such as "ACTIVE". More generally, this tool takes the counterexample output from nuXmv as input, filters its trace for lines matching custom-defined regular expressions and generates an output file containing only lines with matches.

Listing 4.17 shows an excerpt of the counterexample trace, which was the result of filtering the original trace from nuXmv (containing around 700 lines) with our tool having filtered out all lines that do not contain the string "ACTIVE" for the Car process, more precisely the regular expression `.*CarProcess.*.ACTIVE`, e.g., the entries in states from 1.6 to 1.33. For illustration purposes, the listing shows two of the lines filtered out at State 1.4 in gray.

Listing 4.17: Filtered Output of a Counterexample (Listing 9 in [47])

```
Spec:( G !olc.ERROR &  F ( G terminator.state = terminated))    is false

——> State 1.1 <——
  olc.OLC_Car_StateMachineobj.Battery_Level = Initial
  mainAct.Call_Car_Process.SUB_ACTIVE = FALSE
  mainAct.Call_Car_Process.InitialNode.ACTIVE = FALSE
  mainAct.Call_Car_Process.Send_Charging_Finished.ACTIVE = FALSE
  mainAct.Call_Car_Process.Merge.ACTIVE = FALSE
  mainAct.Call_Car_Process.ActivityFinal.ACTIVE = FALSE
  mainAct.Call_Car_Process.Start_Charging_Battery.ACTIVE = FALSE
  mainAct.Call_Car_Process.Merge1.ACTIVE = FALSE
  mainAct.Call_Car_Process.DecBatteryFull.ACTIVE = FALSE
  mainAct.Call_Car_Process.DecVoltageSupplied.ACTIVE = FALSE
  mainAct.Call_Car_Process.Stop_Charging_Battery_2.ACTIVE = FALSE
  mainAct.Call_Car_Process.Stop_Charging_Battery.ACTIVE = FALSE
  mainAct.Call_Car_Process.Merge2.ACTIVE = FALSE
  mainAct.Call_Car_Process.Initiate_Process.ACTIVE = FALSE
  mainAct.Call_Car_Process.Merge3.ACTIVE = FALSE
  mainAct.Call_Car_Process.DecVoltageSupplied1.ACTIVE = FALSE
```

```
   mainAct.Call_Car_Process.DecCableDisconnected.ACTIVE = FALSE
———> State 1.2 <———
   olc.OLC_Car_StateMachineobj.Battery_Level = 0
———> State 1.3 <———
———> State 1.4 <———
   mainAct.Call_User_Process.InitialtoCall_Connect_Cable_Activity. forwardSignal = TRUE
   mainAct.Call_User_Process.Initial.state_tokengenerator = S2
———> State 1.5 <———
   mainAct.Call_Car_Process.SUB_ACTIVE = TRUE
   mainAct.Call_Car_Process.InitialNode.ACTIVE = TRUE
———> State 1.6 <———
— Loop starts here
———> State: 1.33 <———
```

Using this filtered counterexample trace, we found that the Action in the Car Process never changes and that the process remains in the *Initial Node*. In the trace this can be see by *mainAct.Call_Car_Process.InitialNode.ACTIVE* becoming and staying TRUE. In principle, this could be due to different reasons, such as a wrong transition in an object life cycle or an erroneous pre- or postcondition. We continued by investigating pre- and postconditions of the corresponding Action and added an additional filter for the values of the constrained attributes, i.e., the battery level (*olc.OLC_Car_StateMachineobj.Battery_Level*) filtered by the regular expression `.*.Battery_Level`. The resulting trace showed that the battery level is not changing either, which is a strong indication of an erroneous precondition. This problem was confirmed by a manual check of the process and the *Initial Node* action. Removing the wrong precondition introduced via copy&edit, solved the problem.

The successful verification run (for checking our two consistency properties) on the final model took about 11 seconds on the laptop computer specified above. Even though bounded model checking was used, the model was fully verified as nuXmv allows for identifying bounds based on changes that occur in the model. After reaching a bound where no more alterations to the model are possible, which in this case was 37, the model checker stops and shows the verification result.

4.10 Related Work

Now let us compare our approach to related work[3]. First, we refer to work on Activity diagrams, and subsequently on object life cycles. Finally, we compare with work related to semantic action specification.

4.10.1 Activity Diagrams

Related work on model-checking of Activity diagrams can be summarized as follows. Li et al. [58] presented an approach for the model-checker tool SPIN, where an Activity diagram is transformed into the input language of SPIN PROMELA. First, Activity diagrams are transformed to *extended hierarchical automata* (EHA), which are then transformed to PROMELA code. Although our approach also transforms Activity diagrams to automata,

[3]This section is directly taken from [47, Sec. 11]

the mapping of an action is completely different. In addition, neither object life cycles nor semantic specifications are involved, which our approach checks for consistency.

Muram et al. [63] proposed a workflow where both high-level and low-level models are developed. From the high-level model, the temporal logic properties to be checked are derived. The low-level model is translated to the input language of the model-checker tool NuSMV. Model-checking then reveals whether these two models are consistent or not. This is different from our approach, which checks a process model for consistency with object life cycles, which are connected through semantic action specifications.

Robena et al. [44] used UML Activity diagrams for the purpose of specifying a logic controller. They are the basis for two models, an NuSMV model to be model-checked and a VHDL model. However, neither object life cycles nor semantic specifications are involved for consistency checking like in our approach.

Eshuis and Wieringa [38] presented a tool supporting verification of UML Activity diagrams. It transforms Activity diagrams into an input format for a model checker and verifies this input model against propositional requirements given as properties. In contrast to our work, this approach does not utilize pre- and postconditions or object life cycles for checking their consistency with process models.

Building on this work, Eshuis [35] presented an approach for symbolic model checking of Activity diagrams. In particular, a specific property is used to check if the FSM finishes in a defined state. This has inspired our approach for introducing error states. Still, however, neither pre- and postconditions nor object life cycles are taken into account in the verification approach of Eshuis.

4.10.2 Object Life Cycles

Ryndina et al. [73] presented an approach to checking BPMs against object life cycles of artefacts. Each artefact that a BPM operates on is represented by a corresponding object life cycle, both a given one and another one automatically generated from the BPM. Consistency between the BPM and the given object life cycle is checked indirectly by formally comparing the latter with the automatically generated object life cycle. Eshuis and Van Gorp [36] even synthesize hierarchical state machines that can have parallelism, loops, and cross-synchronization. In contrast to our approach, neither Ryndina et al. nor Eshuis and Van Gorp take different life cycle views into account, and neither considers specific contexts in which a process is enacted. In particular, they did not connect a given process model with a given object life cycle using semantic action specification.

Estañol et al. [39] proposed an approach to checking UML models defined through Activity diagrams and state machine diagrams, making use of object life cycle (OCL) for defining operation contracts. In contrast to our approach, they additionally need UML class diagrams for their verification approach. It is completely different from ours by translating all these models together into first-order logic. In order to check behaviors (in the sense of state changes over time), their approach must explicitly define time steps, while time logic is already dedicated to this purpose. Using time logic together with model-checking techniques and tools is well established for general proofs like the ones in our paper. In contrast, the approach in [39] defines certain *tests*, which are implemented by applying existing satisfiability checking tools to the translated models in first-order logic.

Meyer et al. [62] define “weak conformance” between process models and synchronized object life cycles. Their algorithm for soundness checking verifies whether each time an Activity needs to access a data object in a particular state, it is guaranteed that the data object is in or can reach the expected state. In contrast to our approach, they do not use declarative context-dependent action specifications like in our approach.

Generally, our *extended* object life cycle models allow for verification of *non-monotonic* systems. They even allow for modeling communication based on physical interaction in cyber-physical systems. We are not aware of any previous approach to using object life cycles that would have taking non-monotony or cyber-physical systems into account.

4.10.3 Semantic Action Specification

Weber et al. [83] addressed the problem that control flow does not capture what the process activities actually do when they are executed. So, they annotated individual activities with logical pre- and postconditions, specified relative to an ontology with axioms of the underlying business domain. This allowed them to verify the overall process behavior, but they did not utilize semantic action specification in the context of model checking as our approach does.

Based on pre- and postconditions of activities of a process model, Borrego et al. [25] used constraint programming to statically check correctness. In contrast to our approach, they did not include an explicitly defined object life cycle that a process is verified against. Our approach defines the semantics of the pre- and postconditions formally through grounding them in the extended object life cycle.

4.11 Discussion

While designing the workflow with the help of our running example and performing the case study, we learned a few lessons.

Although the automated consistency verification helps find related problems in the models, it is no replacement for domain experts validating the process. However, it can save domain experts precious time, which may be a scarce resource in practice. We also argue that the validation is easy to perform since domain experts are not distracted by consistency problems from the actual validation task. However, this is an interesting question for future studies [47].

Even a skilled modeler may struggle to create formally correct processes. The case study provided suggestive evidence that automated verification helps to improve the models by revealing consistency issues. The generated counterexample also helps to spot the underlying reason for the violation of consistency creation. However, how to fix that issue is to the modeler of the process since this often is more an issue of how to design the process than just fixing a mistake made during modeling the process [47].

It is challenging to interpret the counterexamples as listed by the nuXmv tool. We actually ended up implementing a simple tool that allows filtering the output of the model checker for various properties and enables showing all states in their entirety, which made the process of identifying errors through counterexamples much easier. However, using this tool needs deep knowledge of how model elements are translated into elements of the nuXmv model, especially the naming conventions used. This is not beneficial for an easy use of our approach. Therefore, an automatic translation of the counterexample into a visible trace in the UML Activity diagram would be even more beneficial. However, this is future work [47].

One may ask why we use two different properties to check the two different criteria defined in Section 4.8 for verifying the consistency. At first, we were tempted to use only criteria 1 (and the property associated with it). However, we found that an OLC reaching its error state does not necessarily prevent a process from continuing its operation, e.g., if no *pre-* and *postconditions* are set.

Process models, particularly UML Activity diagrams, may also include object flows. At the moment, our workflow for verification of process-oriented cyber-physical systems does not support them. Future work could aim to support them.

CHAPTER 5

Formal Verification using Structural Abstraction and Selective Refinement

For verification through model-checking, the complexity of the formal models used is highly critical. A usual approach to address this is through abstraction, and we propose *structural abstraction* of the environment model. Technically, our approach approximates environment objects via a composition of cuboids of the same size called *voxels*. The size of the voxels directly influences the accuracy of the approximation. Smaller voxels approximate objects better than larger ones, where the latter are an abstraction of the former. Since this abstraction does not involve any change in the *behavior* of the robot, we denote it as structural abstraction.

In this thesis, we present an approach that reduces the verification time by *systematically abstracting* the environment model through voxels and by *refining* them locally depending on verification results. This approach has the advantage that the models are coarse first and, hence, fast to verify, and only become more detailed in areas where verification runs of abstract models fail. To this end, our defined verification workflow starts with a representation using small voxels, abstracts it by generating a representation consisting of larger voxels, and incrementally adds the details needed for the actual verification.

The remainder of this chapter is organized in the following manner. First, we provide some background material in order to make the thesis self-contained. Then we present our new approach to selective refinement of structural abstraction. For evaluating its feasibility, we present and explain the results of applying our new approach to verifying a safety-critical robot scenario. Finally, we compare our approach with related work, discuss it more generally, and provide a conclusion.

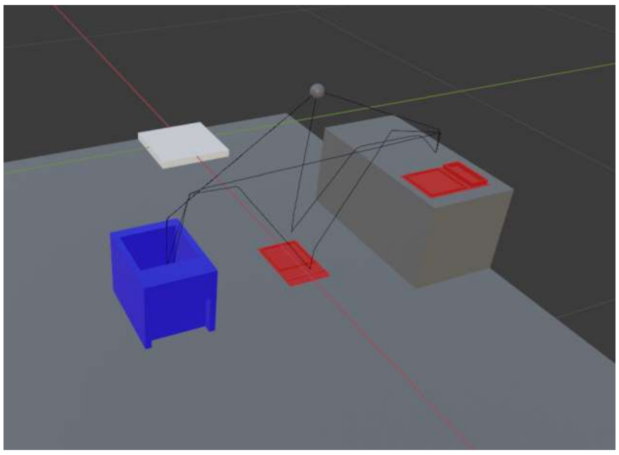


Figure 5.1: Environment model of the running example including gripper (gray sphere) and trajectories (black lines)

5.1 Background

We provide some background material first on our running example of a robot arm performing a pick-and-place task. Subsequently, an existing methodology for verifying such robot applications is described. Since voxels and voxel grids play a major role in this chapter of the thesis, we give some background on them as well. Finally, we introduce *counterexample-guided abstraction refinement* (CEGAR), a methodology to systematically abstract and refine behavioral models.

5.1.1 Running Example

For this chapter, we reuse the running example presented in Rathmair et al. [69]. In order to make it self-contained, we provide a short introduction of this running example here. A robot manipulator with a gripper mounted on the white base plate in the background of Figure 5.1 performs multiple pick-and-place tasks, i.e., picking two objects at their initial position and placing them at different target locations.

At the start of the pick-and-place operation, a large object and a smaller one, which are both not shown in the figure, are located on the red tray in the middle. The first task of the robot is to pick the large object and transfer it to the red tray located on the right. After placing the object on the tray, the robot moves back to the middle tray and picks a second smaller object, which is transferred to the blue box, where it is dropped into. After doing so, the robot arm moves back to the initial position, where it stays still for a moment. This is because the robot performs this application in cooperation with a human being, who manipulates the larger object during the time the robot moves the smaller object. After the human finishes his or her task, the robot picks the object from the tray, drops it into the blue box, and moves back to the initial position. This concludes a cycle of the application, and a new cycle may start.

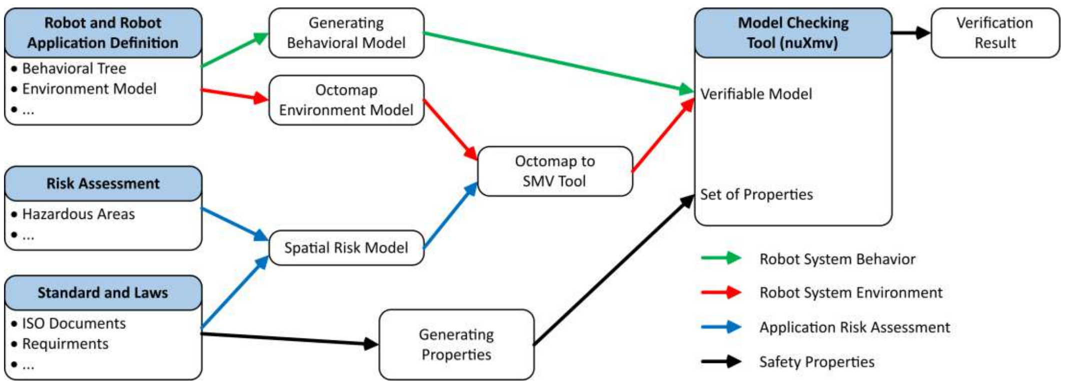


Figure 5.2: Simplified version of the workflow presented in Rathmair et al. [69] (adopted from [69, Fig. 1])

We use this running example for verifying that no collision between the robot and its environment occurs. While verifying this scenario, we use a fixed trajectory with fixed positions of the robot. This is aligned with Rathmair et al. [69], where also the exact coordinates for the initial position and each position on a trajectory from the initial to the end position are used. Our proposed workflow operates on the environment representation only and, therefore, solely structural abstraction is performed.

5.1.2 Verification Methodology for Robot Applications

Rathmair et al. [69] defined a generic verification approach for robot applications. A simplified version of their approach, which leaves out some details not relevant in the context of structural abstraction, is illustrated in Figure 5.2.

Each block represents a process, each comprised of various model generation and transformation steps [69]. The arrows represent data flow between those blocks and are color-coded according to the legend of the figure. Each path deals with a different aspect of the robot application or data needed to verify the application. The presented approach distinguishes between four categories of paths through the blocks.

The green highlighted path (*Robot System Behavior*) category includes data flows containing models, which describe the behavior of the robot system. The red path (*Robot System Environment*) deals with models containing information about the environment in which the robot operates. Data flows containing data associated with risks are shown in blue (*Application Risk Assessment*). Lastly, the black path (*Safety Properties*) is the flow of data related to properties throughout the approach. However, not all paths are entirely independent of one another. For example, data of the risk assessment path is incorporated into the *Robot System Environment* path by the process *Octomap to SMV Tool*. The same applies to the model-checking tool nuXmv, which takes behavioral models, models of the environment and properties, and outputs a verification result for each of the properties.

From a technical viewpoint, the (semi-)automatic approach generates the input model – modeling both the behavior of the robot and the robot's environment – of the model-checker in the SMV modeling language. Additionally, it guarantees that both parts of the model fit together with each other and the properties, at the point of verification. This is mainly done by using unified names for variables that represent the interface between the behavioral and environment model.

5.1.3 Voxel Grid

A *voxel grid* is a construct used in 3D computer graphics, which represents a particular 3D space in terms of its properties. To accomplish this, the voxel grid is composed of individual voxels, which represent a value in the voxel grid, that are all of the same size. Instead of explicitly giving the position of the voxel in terms of 3D coordinates, the position is given relative to other voxels by indexing them. The 3D coordinates of a voxel are calculated using its x, y and z-index, the 3D space the voxel grid covers, and the resolution of the voxel grid.

One way to build a voxel grid is to first define a cuboid (in 3D space) that it should represent. This space is then divided into smaller cuboids that are represented by voxels. The number of cuboids (voxels) is defined by the resolution of the voxel grid and is given via the resolution along each axis, e.g., $4 \times 8 \times 4$ divides the large cuboid into ($4 \times 8 \times 4 =$) 128 smaller cuboids (voxels). Although different resolutions for each axis are possible, it is more common to use the same value for each axis, which is also a power of two, e.g., $4 \times 4 \times 4$ or $8 \times 8 \times 8$. Using this definition, each voxel of a grid with a certain resolution (e.g., $4 \times 4 \times 4$) can be seen as a composition of eight grid voxels with the subsequent higher resolution (e.g., $8 \times 8 \times 8$).

In general, a higher resolution leads to a better approximation of the space, i.e., as more details can be represented, where a particular property holds or not. Figure 5.3 shows the approximation of a sphere in different resolutions and indicates that higher resolutions are more precise. In this example, a voxel is filled in when its value is $True$, i.e., when the sphere at least partly occupies the voxel. Therefore, it results in an approximation that guarantees to enclose the whole sphere, independently of the resolution of the voxel grid. However, other forms of determining the value of the voxel are also reasonable, e.g., setting the voxel value to $True$ only if the entire voxel is part of the sphere. It entirely depends on the particular use case.

Usually, the voxel values are determined based on another form of representation of the 3D space. In our case, the sphere in the left part of Figure 5.3 was modeled in Blender [41] and exported to a file in the *Standard Triangle Language (STL)* file format. We then used tool *binvox* [9] to generate the voxel grids based on the STL file, each with a different resolution. Finally, we used the tool *viewvox* [12] to view the generated voxel grids and capture the views given in Figure 5.3. However, the approach presented in this chapter is not limited to these tools. For example, details of the 3D space could be captured as a point cloud as well, which then is used to generate a voxel representation.

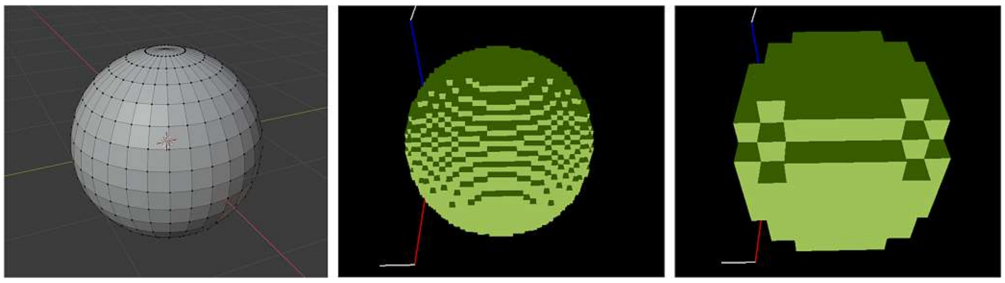


Figure 5.3: Voxel representation of a sphere in different resolutions

left: sphere modeled in Blender
middle: resolution $32 \times 32 \times 32$
right: resolution $8 \times 8 \times 8$

5.2 Selective Refinement of Structural Abstractions

In this section, we present our approach and show how it integrates with the methodology of Rathmair et al. [69]. We start with motivating an approach for the structural abstraction of voxel grids as environment representation. Then we present our approach for generating abstract voxels out of more concrete ones. Using this abstraction process for voxels, we define our CEGAR-inspired verification workflow using structural abstraction and selective refinement. The section concludes with how the workflow is integrated into the verification methodology of Rathmair et al. [69].

5.2.1 Why Structural Abstraction and Selective Refinements Matter

Structural abstractions and their refinements matter because the environment model may become too complex to verify the overall model efficiently. This is also confirmed by analyzing the verification times of different voxel grid resolutions.

Table 5.1 shows the running times needed to verify our running example with different resolutions, i.e., to find a counterexample in this case. The verification times range from a fraction of a second to $\sim 50\,h$, depending on the resolution of the voxel grid. The verification times needed for each individual model-checker run increase rapidly for higher resolutions, since the number of voxels increases with the power of 3. Not only do the verification times differ, but also the lengths of the counterexamples. For instance, the verification with a voxel grid of resolution 2 ($= 2 \times 2 \times 2$) finds a counterexample of length 1 (the property checked is violated in the initial state). With a resolution of 4, a counterexample of length 7 is found. And there is a strong increase in the time needed for performing the verification for increasing resolutions. For a resolution of 128, it took the model-checker a couple of days to finally crash.

As shown in Figure 5.4, for representing the environment of a robot application, using lower resolution – which means a shorter verification time – comes with the drawback of losing details. Whereas the higher-resolution (middle) voxel grid captures the top

Table 5.1: Results without Selective Refinement

Resolution	2	4	8	16	32	64	128
Time	0.1 s	0.2 s	0.4 s	12.2 s	∼19 min	∼50 h	-
Length	1	7	7	7	21	80	-

Resolution ... Resolution of Voxel Grid
Time ... Running Time of Verification
Length ... Length of Counterexample

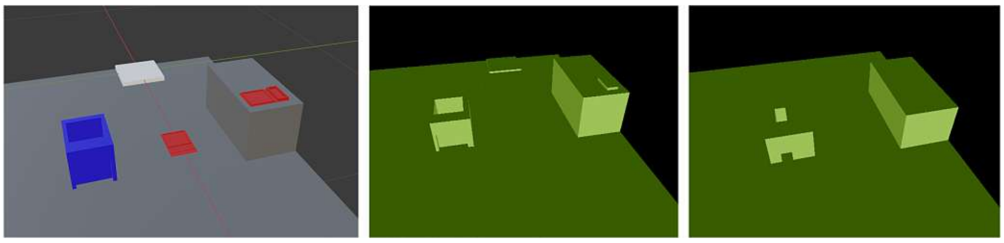


Figure 5.4: Voxel representation of a table with additional installations

left: modeled in Blender
middle: resolution $128 \times 128 \times 128$
right: resolution $32 \times 32 \times 32$

opening of the blue box quite well, the lower-resolution voxel grid does not. However, such details may be important when it comes to deciding if a collision occurs. For example, when using the voxel grid with resolution $2 \times 2 \times 2$, the model-checker already detects a collision with the robot in its initial position. When using a resolution of $4 \times 4 \times 4$, no collision is detected at the same position. This effect of detecting a collision with a specific resolution that disappears with a higher resolution, is also the reason for the different counterexample lengths in Table 5.1.

Unfortunately, one does not know which details of the environment and, therefore, which resolution are needed to verify a property. Verification engineers can pick a resolution based on their experience or perform a detailed analysis of the whole robot application. In general, a higher resolution is preferable, since it means a more realistic representation of the environment than a lower resolution. However, if the resolution is higher than needed, the model-checker run takes longer than necessary. If the resolution is too low, the verification may only fail due to the abstraction and the verification engineer has to determine if this is the case or rerun the verification with a higher resolution. A heuristic may be to use a resolution first that has verification runs in the order of a few minutes, and only increase it when necessary.

Using (automated) structural abstraction in combination with selective refinements as proposed below allows mitigating or even overcoming those problems by using a voxel grid of low resolution first (automatically), thus reducing the time of verification,

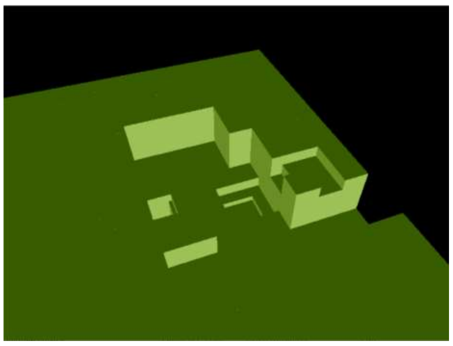


Figure 5.5: Environment representation with more details added by selective refinement

and *selectively* refining the voxel grid at points of particular interest to avoid finding counterexamples that only exist due the to low resolution used.

Figure 5.5 illustrates how selective refinement of a voxel grid with a resolution of $4 \times 4 \times 4$ may be used to capture more details. As one may notice, this representation captures more details for certain parts of the environment, e.g., the opening of the box, than the voxel grid with a resolution of $32 \times 32 \times 32$.

However, there is the challenge to determine where to refine the environment representation. This problem is specifically addressed by our new approach.

5.2.2 Abstracting Voxels for Guaranteeing Over-approximation

As already stated in Section 2.4, over-approximation guarantees that a property holding on the abstract model also holds on the concrete one. However, Counter-example Guided Abstraction Refinement (CEGAR), defined by Clarke et al. [31], is defined only for behavioral models. In this section, we show how voxels can be abstracted to lower the resolution of the voxel grid in such a way that over-approximation is guaranteed.

Before defining over-approximation for voxels, we must better understand it in the context of behavioral models. Over-approximation allows the same or more behavior. Consequently, the model-checker is more likely to encounter states where the atomic prepositions x of a property like $\varphi = AG(x)$ becomes false. Therefore, it is also more likely that the entire property becomes false, which is detected by the model-checker.

Assuming that a voxel can be either (partly) occupied by an obstacle – SOLID – or free of an obstacle – not SOLID – one possible atomic proposition to check whether a collision occurs in a certain step is:

- α = voxels visited by the robot are **not** SOLID

To check if the entire robot application is collision-free, the corresponding property is $\varphi = AG(\alpha)$, i.e., that the atomic proposition does not evaluate to false in any step.

Considering that the approximation does not alter the behavioral part of the model (the robot still moves along the same trajectory given in numerical coordinates), using different (structural) abstractions does mean checking which parts of the 3D space are visited by the robot in a different granularity (resolution). For example, this means checking $4 * 4 * 4 = 64$ voxels instead of $8 * 8 * 8 = 512$ voxels if they are visited and SOLID. However, to guarantee that all violations (collisions) detected by the finer-grained analysis are also detected by the coarser-grained one, we must set the voxel values right.

Assuming we have two voxels, a free one and an occupied one named v_0 and v_1, respectively. v_0 has the value False (since it is not SOLID), and v_1 has the value True (since it is SOLID). For reducing the number of voxels by joining them together into one abstract voxel v_a, it must be defined which value is assigned to that voxel. The abstract voxel v_a needs to have the value True, to fulfill over-approximation.

Considering an abstract voxel as SOLID when at least one (less abstract) voxel it is composed of is SOLID also corresponds nicely with the output of the tool *binvox*. Given the same STL-file, *binvox* generates voxel grids in such a way that an obstacle occupies more space in a lower-resolution voxel grid than in a higher-resolution one. Actually, we used voxel grids of different resolutions exported by *binvox* to check this approach and to verify the implementation of the abstraction mechanism.

In general, however, the abstraction approach depends on the property to be checked. Consider the following property:

- $AG(voxels\, visited\, by\, the\, robot\, are\, FREE)$

Assume that a value of True means that the voxel is FREE. Then the abstraction mechanism of setting the value of an abstract voxel to True when at least one of the more concrete voxels has a value of True does *not* fulfill over-approximation.

5.2.3 Our Verification Approach using Selective Refinement of Structural Abstractions

The verification approach presented in this section uses structural abstraction of voxel grids in combination with selective refinements of individual voxels. It is inspired by the ideas behind CEGAR as we also perform an initial abstraction and improve the abstract model in incremental steps using information gathered by analyzing counterexamples.

Our workflow, as illustrated in Figure 5.6, consists of multiple steps. First and foremost, there is an initial step of abstracting the voxel grid provided to the workflow. After that, the whole workflow mainly operates on a voxel grid with reduced (compared to the provided one) resolution. While the provided higher-resolution voxel grid is still available during the whole workflow, it is not directly used for verification.

After the initial abstraction is performed, the verification loop, consisting of performing a verification run, analyzing the counterexample and refining the environment model, starts. Details on those steps are given below.

The workflow ends if either the verification run does not detect a violation of the property (and, therefore, does not generate a counterexample) or the proposed refinement is considered not valuable or even impossible. In the first case, the workflow ends with the result that the verification has passed. In the second case, the verification fails and the workflow provides the counterexample that caused it.

During the execution of the workflow, we distinguish between three resolutions of a voxel (grid). Those are:

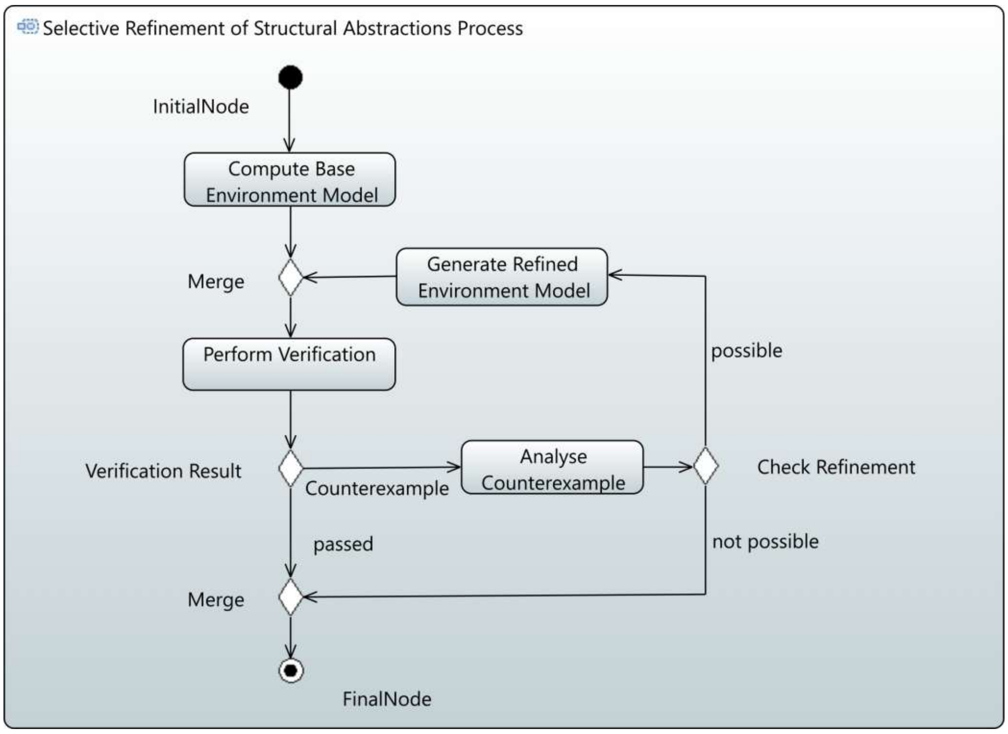


Figure 5.6: Verification Workflow with Selective Refinement of Structural Abstractions

- **Max-resolution:** The highest resolution used during the execution of the workflow. This is the resolution of the voxel grid stored as *binvox*-file.
- **Base-resolution:** This resolution is chosen by the verification engineer and is the resolution the voxel grid is reduced to at the start of the workflow. It represents the coarsest resolution that is used during the execution of the workflow. It is the resolution of the initial abstraction.
- **Voxel-resolution:** This is the specific resolution of a particular voxel and is neither greater than Max-resolution nor smaller than Base-resolution.

To understand the difference between those resolutions better, we give a short example. The voxel grid provided to the workflow has a resolution of 128 ($128 \times 128 \times 128$) and, therefore, divides the space covered by the voxel grid into $2,097,152$ individual voxels. Hence, the Max-resolution of the workflow is 128. The initial abstraction is set to generate a voxel grid with a resolution of 4 ($4 \times 4 \times 4$), i.e., the Base-resolution of the workflow is 4. Note, with its 64 voxels, this voxel grid covers the same space as the $2,097,152$ voxels provided to the workflow. Thus, those 64 voxels have $32,768$ times the size of the voxels provided to the workflow. However, instead of explicitly spelling out the size of voxels, we just say that a voxel has a particular Voxel-resolution. In our example, all voxels in the voxel grid provided to the workflow have a Voxel-resolution of 128 and all voxels of the voxel grid generated by the initial abstraction have a Voxel-resolution of 4. After introducing refinements, the environment representation does have voxels of various sizes and, therefore, various Voxel-resolution.

As already stated, the first step of our workflow is to generate a voxel grid with reduced resolution. Depending on the Max-resolution, the preset Base-resolution and the property to be checked, an abstraction according to our approach for abstracting voxels (Subsection 5.2.2) is done. In our example, this step determines the value associated with a voxel of resolution 4 by joining $32,768$ voxels of resolution 128 together. All the 128-res voxels a 4-res voxel is composed of are joined. For example, the 4-res voxel with index $x_4 = 0$, $y_4 = 1$, $z_4 = 0$ is composed of all 128-res voxels whose index fulfills the following expressions $x_{128} \in [0, 31]$, $y_{128} \in [32, 63]$ and $z_{128} \in [0, 31]$.

The first step in the verification loop executes the action *Perform Verification*. During this step, all the models required (like environment model, behavioral model, etc.) are put together to form the overall model. This model is then handed over to the model-checker to be verified against the given property. The output of the model-checker is the result of this step and is either the statement that the property is verified successfully, or a counterexample, which shows that the property is violated. Depending on this output, the workflow either stops with the result of successful verification of the model against the property, or it proceeds with executing the next action *Analyze Counterexample*.

The action *Analyze Counterexample* analyzes the output of the previous verification step. In our implementation, the output is captured as a log file and, hence, the file is parsed and the (human-readable) counterexample documented in it is analyzed. The counterexample contains information on how variables of the model change at each time step of the verification run. With this information and knowing the names of the variables used to represent the index of a voxel, the specific voxel that caused the property violation and the time step when the violation occurred can be determined. Based on this information, the action then outputs a suggestion for voxels to be refined. In our implementation, the suggestion is the specific voxel causing the violation. However, as we explain in the discussion section below, a more elaborate suggestion containing multiple voxels may be possible.

The suggested refinement can only be performed if the voxel has not Max-resolution already. Otherwise, no further refinement is possible and the workflow ends with the result of a failed verification.

The actual refinement is performed in the *Generate Refined Environment Model* action. In this step, the (abstract) voxel is refined into eight new voxels with the next higher resolution, e.g., a voxel of resolution 8 is refined into eight voxels of resolution 16. To determine the Max-resolution voxels to be combined to form a particular new voxel, the position of the voxel to be refined and the position of the new voxel inside of the refined one are used. To compute the value (SOLID or not) for each new voxel, the identified Max-resolution voxels are combined via our approach for abstracting voxels.

Finally, the whole environment – consisting of the voxel grid in base resolution and refined ones – is exported to its nuXmv representation.

The newly generated environment representation is then used during the next *Perform Verification* step of the workflow. The cycle of verification, analyzing the counterexample,

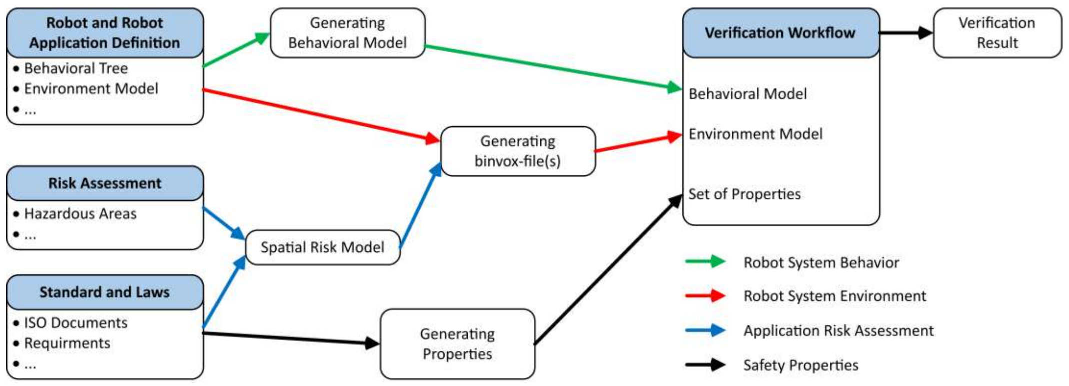


Figure 5.7: Adopted version of the methodology to incorporate our workflow

and generating a new environment representation goes on until the workflow terminates.

5.2.4 Integration into Preexisting Methodology

Figure 5.7 shows how the proposed workflow is integrated into the preexisting methodology of Rathmair et al [69].

Compared to the original methodology, only minor changes were necessary. The process of verification by the model checker nuXmv is simply replaced by the tool implementing the verification workflow. In addition to that, adopting the *Robot Environment Path* was also necessary, since the original methodology would hand over a SMV model of the environment. However, our workflow asks for binvox-files representing the environment. Therefore, both processes of the *Robot Environment Path* were replaced by a process generating the needed binvox-file(s).

5.3 Results

In this section, we present and explain the results of applying our new approach to verifying the robot scenario. For demonstrating that our approach can improve model-checking performance, we verified this scenario using our workflow with a Max-resolution of 128 first. Note again, that there were no results for verification without selective refinement for this high resolution, since the model-checker did not finish with a result even after a couple of days. Table 5.1 above shows that even for a lower resolution of 64, ~50 h were needed for finding a counterexample.

Table 5.2 shows the results for our approach with selective refinement, for various values of Base-resolution, which has to be set before each verification run as a kind of parameter. A Base-resolution of 128 does not make sense for a Max-resolution with the same value, since using a Base-resolution equal to the Max-resolution leads to the same model-checker run as without selective refinement. Hence, there is no related result given in this table. Also the verification with a Base-resolution of 64 took extremely long, but

Table 5.2: Results with Selective Refinement – Max-Resolution 128

Base-Resolution	2	4	8	16	32	64
Time	∼4 min	∼3 min	∼3.5 min	∼14 min	∼6 h	∼8.4 d
Length	80	80	80	80	80	80
Refinements	39	35	32	23	13	3

Base-Resolution ... Base-Resolution used in the Workflow
Time ... Running Time of Verification
Length ... Length of Counterexample
Refinements ... Number of Refinements made

using such a high Base-resolution is not reasonable since it constrains our approach too much.

The lowest running times in the order of a few minutes have been achieved with low values for Base-resolution, i.e., when starting the model-checker runs with low resolutions, which finish fairly quickly. After that, selective refinement shows its benefits by exploring higher resolutions only where necessary for finding a real counterexample. Hence, with lower values of Base-resolution, a (real) counterexample for Max-resolution of 128 was possible to be found within a few minutes, while the model-checker directly running with the high resolution and without selective refinement did not finish with a result even after a couple of days.

For the purpose of illustrating selective refinements, Figure 5.8 visualizes an example of the evolution of the voxel grid when applying the workflow for performing refinements with a Base-resolution of 4 and Max-resolution of 128. The leftmost subfigure shows the environment in a resolution of 4. Neither the blue box nor the raised tray are visible. After a few refinements, both can already be seen vaguely. With its 35 refinements, the final representation gives even more details. Note, that the opening of the box seems not essential to disprove the property (in this case).

We also investigated the relative performance of our new approach for other values of Max-resolution. Table 5.3 shows the overall results for all combinations of the investigated pairs of Base- and Max-resolutions ranging from 2 to 64 and from 2 to 128, respectively. The running times from Table 5.2 are included here in the bottom row for facilitating comparisons. Note, that also the running times from Table 5.1 are included, in the diagonal of Table 5.3. For all combinations with a Max-resolution of 4 and 8, our approach using selective refinement takes slightly more time, but this is in the order of less than a second and, hence, does not really matter. We suspect that this is due to the workflow's overhead for generating the environment model(s), which is small in absolute terms, but compared to these very short verification times, relatively large. For larger Max-resolutions, this overhead can be neglected. The running times of the model-checker for Max-resolutions 16, 32 and 64 show the same pattern as the ones for

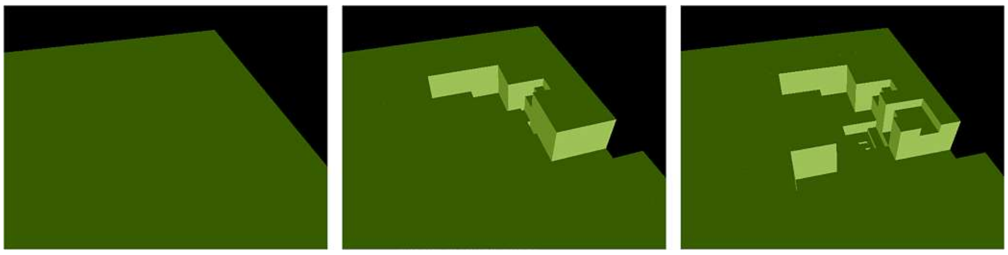


Figure 5.8: Evolution of the voxel grid with Base-resolution 4 and Max-resolution 128

left: without any refinement
middle: with 16 refinements
right: with 35 refinements

128. The minimal running times were achieved for low values of Base-resolution, but not for the lowest one of 2.

Table 5.3: Verification-Times of all Base- and Max-Resolution Combinations

		Base-Resolution					
		2	4	8	16	32	64
Max-Resolution	2	0.2 s	-	-	-	-	-
	4	0.6 s	0.4 s	-	-	-	-
	8	1 s	0.8 s	0.6 s	-	-	-
	16	1.4 s	1.2 s	1.2 s	12.4 s	-	-
	32	7.2 s	7 s	8.5 s	~1 min	~19.4 min	-
	64	~2.7 min	~2.5 min	~2.6 min	~12 min	~5.3 h	~50 h
	128	~4 min	~3 min	~3.5 min	~14 min	~6 h	~8.4 d

Base-Resolution ...	Base-Resolution used during the Workflow
Max-Resolution ...	Max-Resolution used during the Workflow

To get a better understanding of the dependencies between the values chosen for Base-resolution and the resulting total times needed for the verification through our selective refinement approach, we further analyzed the information gathered during the execution of the workflow. Figure 5.9 compares relevant data of the different verification runs with Max-resolution 128 and Base-resolutions ranging from 2 to 16. While the workflow was started with different Base-resolutions, all these executions found the same collision as the counterexample, more precisely in terms of the position of the robot arm. These charts depict only the total execution times needed by the model-checker. Therefore, the total times are slightly shorter than the times given in the column *Time* of Table 5.2. We decided to do so to eliminate any influence that our prototypical implementation of the workflow may have.

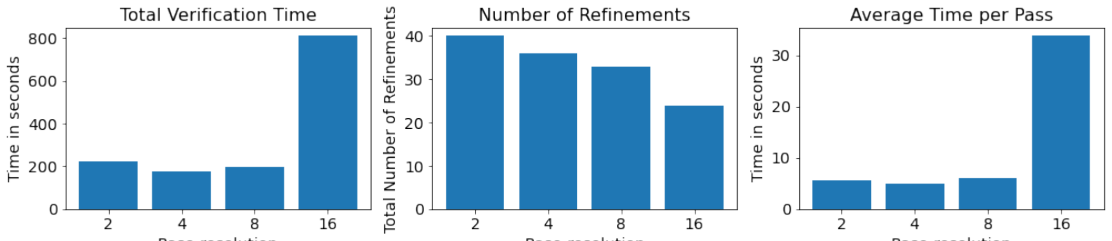


Figure 5.9: Comparison of more detailed results for different Base-resolutions and Max-resolution 128

The leftmost diagram of Figure 5.9 visualizes the total time in terms of the sum of the *Perform Verification* actions needed. The chart in the middle shows the number of refinements that were enacted. The rightmost chart shows the average time to perform a single pass in terms of the model-checker run of one refinement. As we can see, not only the *Total Verification Time* has its minimum at a base resolution of 4, but also the *Average Time per Run*. The number of refinements, however, is decreasing with increasing Base-resolution.

In order to get a better understanding, we also had a closer look by comparing the number of refinements with their *depth*. Refinement-depth means how many refinements are enacted in a sequence to reach the desired level of detail. For example, assuming a Base-resolution of 2, a refinement-depth of 1 leads to voxels with a resolution of 4, a refinement-depth of 2 leads to voxels of resolution 4 first and then 8, etc. The diagrams in Figure 5.10 show that increasing Base-resolution does not only lead to decreasing numbers of refinements, but also to decreasing refinement-depths, on average.

The number of refinements with the respective highest depth (i.e., the ones leading to voxels of resolution 128) differs for different Base-resolutions. We suspect that this is due to the implementation of the model-checker, more precisely its part that generates the counterexamples. The model violates several voxels at the same time. However, which voxel is chosen for the counterexample influences the workflow. If the model-checker chooses a voxel that is refined already, the workflow stops and if it chooses a non-refined voxel, the workflow takes another round in the refinement loop. The results are fully reproducible, indicating that the selection was not done randomly.

To further investigate the internal behavior of the approach, we had a closer look into different phases:

1. before finding the real counterexample (of length 80 here),
2. finding the real counterexample, and
3. proving that this counterexample is real.

The three diagrams of Figure 5.11 show the times needed for the model-checker runs in these three phases, for Base-resolutions 2, 4 and 8 (and Max-resolution 128). The relative times needed for the model-checker runs for the various Base-resolutions in the

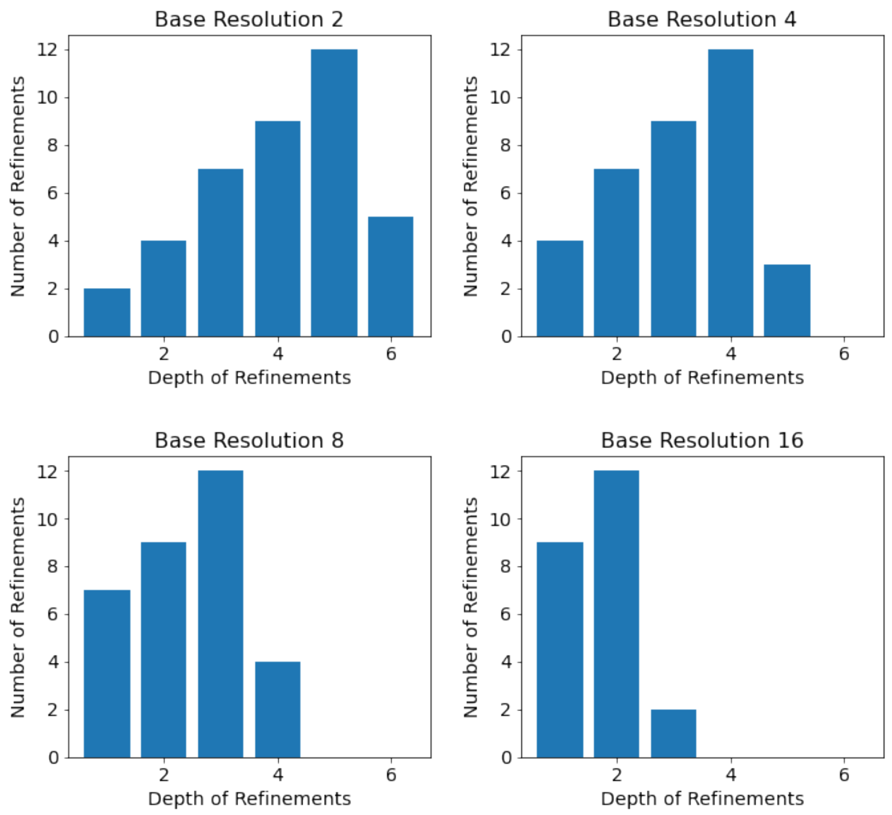


Figure 5.10: Number and Depth of Refinements made during verifying the running example

first two phases show the same pattern as those of the total times in Figure 5.9, where the lowest times result for Base-resolution 4. In the third phase, however, the time needed for the model-checker runs decreases with increasing Base-resolution. Since for lower Base-resolution the refinement-depths are higher, on average, as shown above, we conjecture that introducing a refinement of higher depth costs more in terms of verification time than a refinement of lower depth. The reason may be that with increasing depth, more and more variables (implementing the refinements) became interdependent, so that the SAT problems became harder to solve.

5.4 Related Work

A bounding volume hierarchy is a hierarchy that arranges bounding volumes of objects into a tree structure [34, Chapter 6]. It is often used for collision detection, e.g., in game engines. One way to model a voxel grid as a bounding volume hierarchy is using an octree. A node of an octree has exactly eight children or no children at all. Each leaf node of the octree represents a voxel of the voxel grid. All other nodes of the octree represent what we call abstract voxels of varying sizes depending on the node's level in

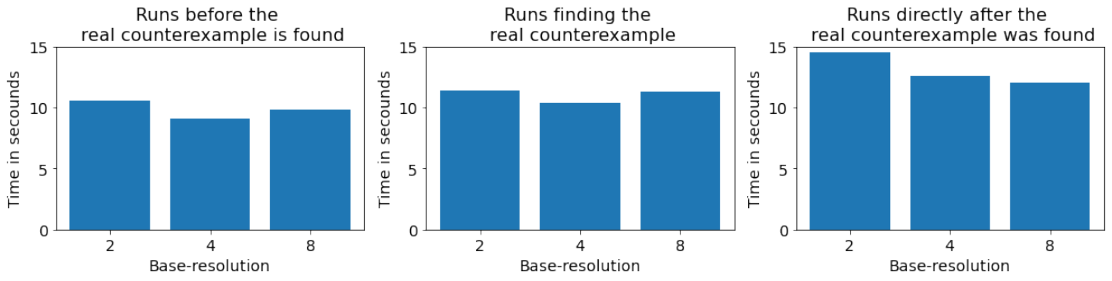


Figure 5.11: Times of the verification runs finding the (real) counterexample with length 80

the octree. However, modeling the whole octree would lead to a very large environment model. That is why our workflow uses environment representations that include only distinct parts of the octree and, hence, the bounding volume hierarchy of the voxel grid, depending on the selective refinements introduced.

Multi-level voxels models [86, 43] may be an alternative to our approach of representing different resolutions. It uses two types of voxels – coarse and fine resolution voxels. The coarse resolution voxels form the base voxel grid. At object boundaries, a coarse resolution voxel is subdivided into fine resolution voxels. This enables a more detailed approximation of objects with a lower overall number of voxels as compared to using fine resolution voxels throughout the whole model. Our approach also refines course voxels into finer-grained ones. However, it selectively refines voxels deemed important for a particular verification task, and it does so on several levels of granularity.

Babić and Hu [21] introduced structural abstraction of software by providing an automatic abstraction-checking-refinement framework. It also follows the general approach of CEGAR of generating a coarse initial abstraction and refining it based on counterexamples. The framework uses "the natural function-level abstraction boundaries present in software"[21] to omit details about function behavior. Initially, their approach treats the effects of function calls as unconstrained variables. Later constraints are added based on the counterexamples. In essence, they use structural information gathered by analyzing the software to guide behavioral abstractions. Our approach differs since it does not abstract the behavior of a system. Instead, it abstracts structure of the environment the system is embedded into.

Fishwick and Lee [40] group the behavior of different (physical) entities in a structure, which is also abstracted in their approach. One state in their finite state automata (FSA) represents a high-level state of the system. The system's behavior in a specific state is then comprised of the behavior of the individual entities during this state. In contrast, we neither use abstract states nor abstract behavior in our approach.

Yang and Chen [85] use artificial intelligence (AI) based shape abstraction to map point clouds into a representation consisting only of cuboids. However, inferring that a concrete model satisfies a property when the abstract model satisfies it relies on overapproximation – in our case, this means that the whole "real" object is contained in its abstract representation. Using their method for abstracting the environment does not guarantee over-approximation, however. Therefore, their approach cannot be used as the basis for generating the voxel grids that we need for our approach of selective refinement.

Rathmair et al. [69] proposed a verification workflow for robot applications that the one here builds upon. To overcome the hassles that come with using a high-resolution voxel grid, Rathmair et al. divide the environment into multiple objects, e.g., table and blue box. Each voxel grid has a different resolution, depending on its size and the details of the object required for the verification. For example, a voxel grid of resolution 16 covering only the blue box has significantly smaller voxels and, therefore, provides more box details than a voxel grid with the same resolution covering the whole environment. Our new approach is different because the level of detail is not homogeneous over a whole object. Instead, the model may provide more details of certain parts of an object when they are relevant, and the selective refinement automatically determines where more details are needed.

Several alternatives to Rathmair et al. [69] have been published on formal verification of applications with human-robot collaboration. Lestingi and Longoni [56] presents a tool-supported model-driven approach based on the "SAFER-HRC" [19] methodology, which enables formal verification of the risk level of collaboration. Askarpour et al. [20] also uses a combination of model-checking and simulation “in order to combine the precision of the former and exhaustiveness of the latter” [20]. This enables the approach to determine if the model checker falsely identified a situation as hazardous.

The recently published paper, [49], used formal verification in combination with simulation to verify robot applications. The approach uses two models, one for formal verification and one for simulation. The more abstract model for formal verification is an over-approximation of the simulation model. A formal verification step first identifies “a set of potentially critical cases which are then closer examined in simulation”[49].

5.5 Discussion

There is a tradeoff between the number of model-checking runs for refinements and the time needed for each model-checking run. Generally, a model-checker run with a lower resolution takes less time than one with a higher resolution. However, workflow executions with lower Base-resolution need more refinements and, therefore, more individual model-checker runs. Based on our analysis of this tradeoff, determining a value for Base-resolution upfront can be done by educated guessing and, later, through experience.

Although our new verification workflow has shown its potential to strongly outperform verification without refinements for our example, we see room for further improvement. Detailed analysis of the individual refinements made while verifying our example showed that generated counterexamples tend to have the same length. That is, the same robot position as in the verification run before causes a counterexample even though a refinement was made. The reason is that a counterexample only highlights one voxel that causes a violation, although, at each robot position, more than one voxel may cause a violation. This may lead to situations where the model-checker finds a violation, a refinement is done, and the next verification step finds a violation at the same robot position. We saw this phenomenon when our approach tries to overcome the counterexample at length 21, when using a resolution of 32 without selective refinement. Manually analyzing this exact robot position revealed that up to 6 voxels – dependent on the Base-resolution – violate the property. However, the workflow refines the voxels one by one, so that up to 5 additional model checker runs may be needed. Static analysis of the exact situation where the counterexample occurs, and not only refining the voxel given in the counterexample could reduce the number of model-checker runs needed and, hence, the verification time.

Our proposed workflow does not check the outcome of the refinement of a voxel for the following situation. A SOLID voxel proposed for refinement may only be composed of SOLID higher-resolution voxels, i.e., voxels with the subsequent higher resolution. In such a situation, it is unavoidable that the model-checker detects a collision in its next run. Refining the higher-resolution voxels at the same time, even over several refinement levels, could skip such model-checker runs. In our robot application, such a situation has been encountered multiple times. With a Base-resolution of 4, already the first refinement has this structure. Introducing multiple refinements in such a situation could have skipped several model-checker runs and, hence, reduced the overall verification time. There can even be the extreme case that a SOLID voxel proposed for refinement is entirely composed of SOLID voxels even at the Max-resolution. In this case, the introduction of new refinements can be skipped since the model-checker is guaranteed to find a counterexample, anyway, leading to reduced verification time.

Although the structural abstraction proposed in this thesis is done on voxel grids only, the overall approach is not limited to them. However, it is necessary to have some representation of the relationship between structural elements on one level of abstraction to elements on another level of abstraction. In the case of voxel grids, the indexes of a voxel allow computing the indexes of the lower-resolution voxel that it is part of and the indexes of all higher-resolution voxels that are abstracted by it. In general, it may be necessary to represent this relationship explicitly.

Finally, since the voxel grid provided to the workflow is already an abstraction of the real environment, it would have to be made sure that it is generated in such a way that this abstraction is an over-approximation of the real environment. Unless this is ensured, however, the workflow may detect a collision that does not actually occur in the real environment. However, this issue is out of the scope of the workflow, and it arises also when directly performing model-checking without abstractions and refinements, of course.

CHAPTER 6

Discussion and Future Work

In this thesis, we presented three different approaches for solving three different problems when it comes to the verification of cyber-physical systems. However, we found that they have one thing in common besides using models and model-checking – the extensive use of counterexamples as a source of information. Counterexamples are used explicitly in the workflows of our top-down design process and the verification workflow for robot applications for generating refinements of abstract models. Additionally, counterexamples may be used to improve the system design. In the top-down design process, “real” counterexamples are used to improve the qualitative model. Therefore, design decisions may base on them. Performing the case study to our verification workflow of process-oriented CPSs, we realized that the counterexample directly influenced the changes made to the model. Hence, counterexamples influence the (process) design here as well.

We already discussed how the order in which the counterexamples are given influences the verification process for robot applications. Similar things can be said when counterexamples guide the system design. Because counterexamples are given on a one-by-one basis and rerunning the verification with the same model gives the same counterexample, one gets a very narrow view of the problem. The narrow view may lead to losing the big picture when fixing the “bugs” of the system. In contrast, running a set of tests gives more than one result and, thus, depending on the test cases, may provide more detailed insight, leading to better fixes. To compensate for the lack of insight when using model-checking, we deem it essential to perform a validation step after making a few changes to (the model of) the system.

As this thesis shows, formal verification and, in particular, model-checking is only feasible when done on abstract models. However, usually, those models are the basis of the implementation of the system. Thus, the challenge of verifying the implementation against those models remains. There exist numerous ways to generate test cases using a model-checker [45, 28, 46, 72, 42]. Future work may explore using such approaches for generating test cases for verifying the implementation against its (abstract) model.

For the entire thesis, we stuck to the model-checker nuXmv[27]. This was partly due to the research team's previous experience with nuXmv and its predecessor NuSMV [66] and partly because it is free of charge for academic use. However, there may be a (commercial) model-checker better suited for each of the approaches presented in this thesis. For example, SPIN [79] allows the definition of processes and their synchronization in various ways, which could be beneficial to the verification of process-oriented CPSs. However, we leave evaluation of different model-checking tools for future work.

In addition, each topic in the main chapters (3-5) has a potential future work on its own.

As discussed in Section 3.8, manually doing the top-down design workflow for larger systems may not be reasonable. Future work could explore the feasibility of automation for the individual steps of the workflow and implement it.

The workflow for verifying consistency between process-oriented models of the CPS and OLC (given in Section 4.2) currently lacks the support for *Object Flows*. Future work may include such support by modifying the predefined nuXmv-Module and the semantic specification.

Our verification approach using selective refinement of structural abstractions (described in Section 5.6) has also potential for improvements and, therefore, future work to be done. We see the potential for improvements when it comes to checking the counterexample and generating the refinement. Future work may evaluate and implement the potential improvements already discussed in Section 5.5.

CHAPTER 7

Conclusion

Formal verification of cyber-physical systems is challenging in many ways due to their complexity. This thesis shows how to overcome some of those challenges by answering the following key research questions:

- How can formal verification help to improve models in a top-down design process of CPS?

  We developed a workflow that provides a systematic way for the top-down design with integrated verification and validation of the models of the CPS and its environment. Our novel top-down approach works iteratively on different levels of abstraction while keeping the models formally consistent. Like in CEGAR, the proof through model-checking is only done on the abstract level (with less complexity than the concrete model). However, in contrast to CEGAR, the purpose is not to verify an existing concrete model but to design both models from scratch. An additionally performed cross-check provides some evidence on the correctness of the workflow itself and of our enactment in the course of this case study.

- How to check consistency between process-oriented CPSs and their environment as early as possible in the course of the development?

  We developed and implemented a fully automated workflow to verify CPSs, based on the novel approach of using process models and object life cycles. It allows automated verification of process-oriented models of the CPS's behavior against a model of the behavior of the environment that the CPS is supposed to operate in. This, enables verification at a very early stage of a model-based development process.

- How to improve the runtime efficiency of the formal verification of certain robot applications using structural abstraction based on a voxel representation?

  In this thesis, we propose a new approach to formal verification of safety-critical robot applications, using structural abstraction of environment models and their selective refinement. While this is inspired by CEGAR, which abstracts and refines behavioral models, our main contribution is to show that major improvements of the efficiency and, hence, applicability of model-checking are feasible even without such changes in a behavioral model.

Overall, we showed how abstraction could be used, in combination with creating, connecting and transforming models, to improve model-checking performance. These improvements, in turn, make model-checking a more viable method for verification.

Finally, the thesis investigates different roles that model-checking can take. One role is model-checking as a tool for improving (models of) the cyber-physical system by uncovering property violations. Another role is model-checking as the subject of improvements, which are done via abstraction and advanced modeling technics.

APPENDIX A

Qualitative Models and their Refinement

This chapter of the appendix gives the final qualitative model and mapping of the case study.

A.1 Qualitative Models

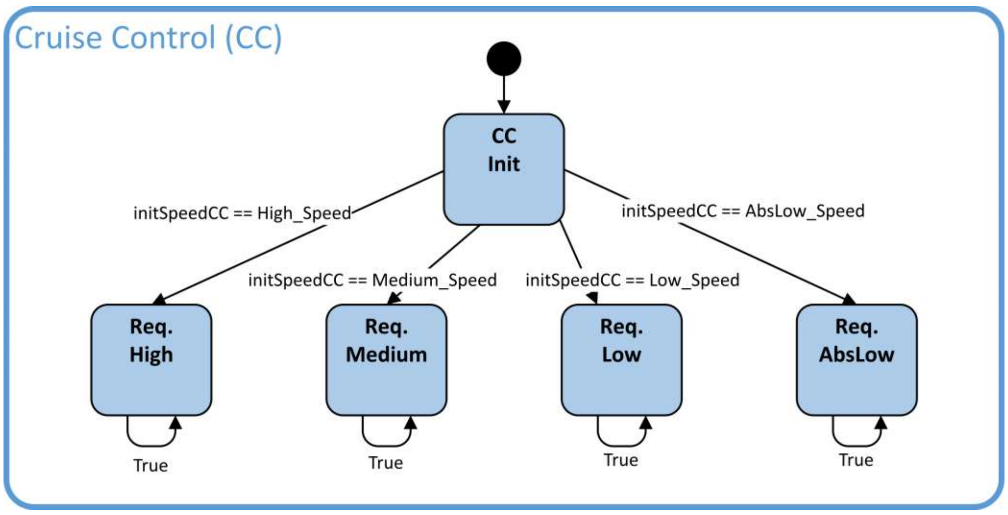


Figure A.1: Final Model of ACC – Cruise Control

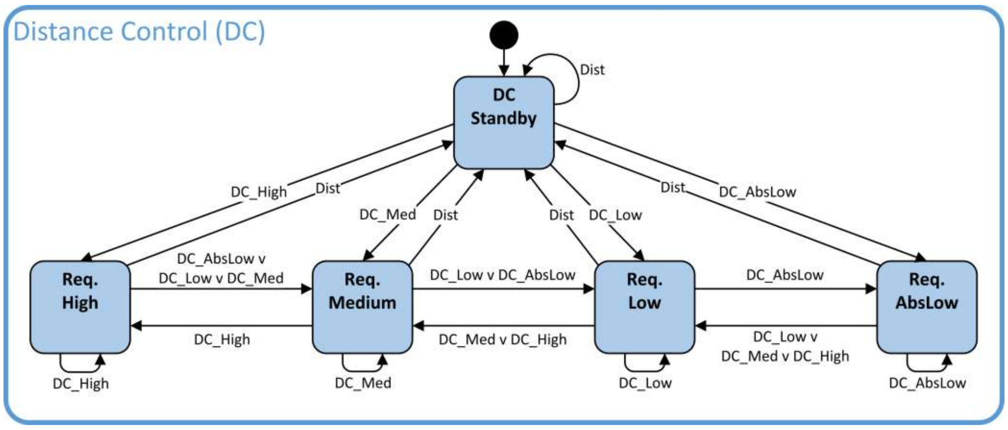


Figure A.2: Final Model of ACC – Distance Control

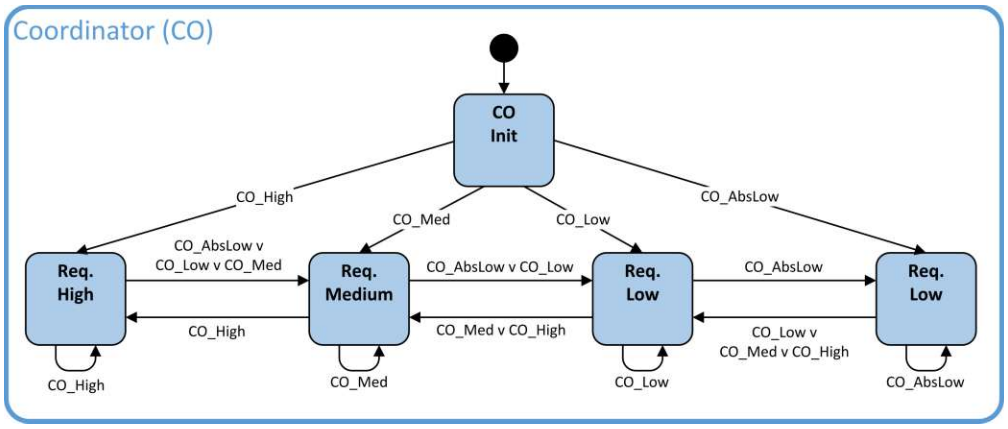


Figure A.3: Final Model of ACC – Coordinator

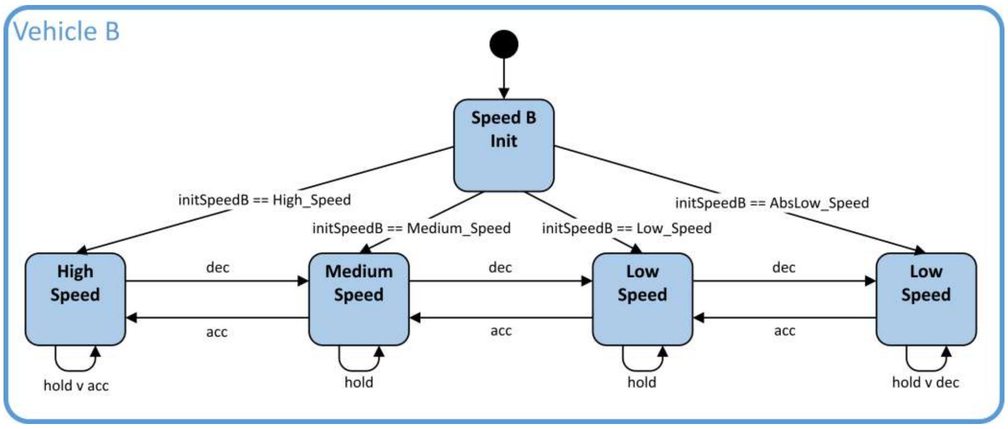


Figure A.4: Final Model of ACC – Vehicle B

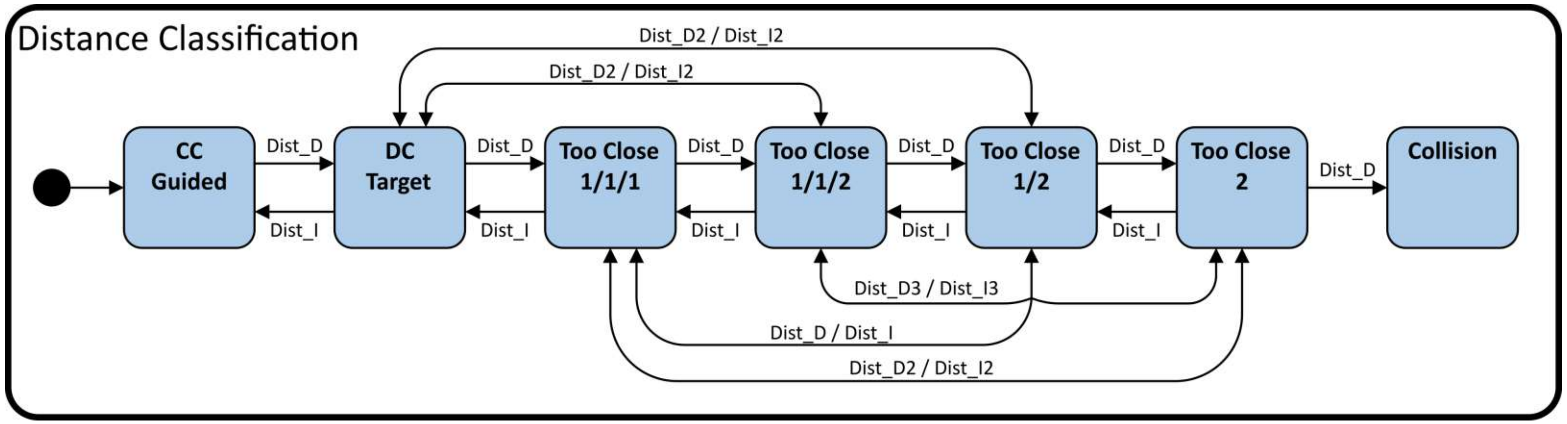


Figure A.5: Final Model of ACC – Distance Classification
Transitions for holding the distance where omitted. Transitions may be traversed in both direction, however, under different conditions. For example, a transition labeled with *Dist_D2*/*Dist_I2* can be traversed from a higher distance to a lower one (left to right) if condition *Dist_D2* holds and from a lower distance to higher one (right to left)if *Dist_I2* holds.

A.1.1 Transition Conditions

Table A.1: Final Model of ACC – Transition conditions of Distance Control FSM given in Figure A.2 (altered version of [60, Table 13])

$\text{Distance}_{\text{del}} \backslash \text{Speed } \text{B}_{\text{del}}$	AbsLow	Low	Medium	High
CC Guided	Dist	Dist	Dist	Dist
DC Target	DC_AbsLow	DC_Low	DC_Med	DC_High
Too Close 1/1/x	DC_AbsLow	DC_AbsLow	DC_Low	DC_Med
Too Close 1/2	DC_AbsLow	DC_AbsLow	DC_AbsLow	DC_Low
Too Close 2	DC_AbsLow	DC_AbsLow	DC_AbsLow	DC_AbsLow

Table A.2: Final Model of ACC – Transition conditions of the Coordinator FSM given in Figure A.3

Req. DC\CC	AbsLow	Low	Medium	High
No Req.	CO_AbsLow	CO_Low	CO_Med	CO_High
AbsLow	CO_AbsLow	CO_AbsLow	CO_AbsLow	CO_AbsLow
Low	CO_AbsLow	CO_Low	CO_Low	CO_Low
Medium	CO_AbsLow	CO_Low	CO_Med	CO_Med
High	CO_AbsLow	CO_Low	CO_Med	CO_High

Table A.3: Final Model of ACC – Transition conditions of Distance Classification FSM given in Figure A.5

Speed A\B	AbsLow	Low	Medium	High
AbsLow	H	H/I	H/I/I3	H/I/I2/I3
Low	H/D	H/I/D	H/I/I3	H/I/I2/I3
Medium	H/D/D3	H/D/D3	H/I/D	H/I/I3
High	H/D/D2/D3	H/D/D2/D3	H/D/D3	H/I/D

H . . . Hold distance I . . . Increase distance D . . . Decrease distance

A.2 Mapping

Table A.4: Final Model - Mapping of qualitative states to quantitative values

	Range/Value
High Speed	$(25\,m/s,\ 30\,m/s]$
Medium Speed	$(20\,m/s,\ 25\,m/s]$
Low Speed	$(15\,m/s,\ 20\,m/s]$
AbsLow Speed	$15\,m/s$
CC Guided	$(50\,m,\ \infty]$
DC Target	$(35\,m,\ 50\,m]$
Too Close 1/1/1	$(25\,m,\ 35\,m]$
Too Close 1/1/2	$(22.5\,m,\ 25\,m]$
Too Close 1/2	$(15\,m,\ 22.5\,m]$
Too Close 2	$(0\,m,\ 15\,m]$
Collision	$[-\infty,\ 0\,m]$

APPENDIX B

Model-Checking of Process Consistency

Listing B.1: nuXmv Module representing the *MainActivity*

```
MODULE MainActivity(controlFlowIn, controlFlowOut, pre, post, container, resetExt
    )
  VAR
    InitialNodetoForkNode_0: ControlFlow();
    ForkNodetoCallDeliveringCompanyProcess_1: ControlFlow();
    ForkNodetoCallCustomerCompanyProcess_2: ControlFlow();
    CallDeliveringCompanyProcesstoJoinNode_3: ControlFlow();
    JoinNodetoActivityFinal_4: ControlFlow();
    CallCustomerCompanyProcesstoJoinNode_5: ControlFlow();

    InitialNode: InitialNode(controlFlowIn.forwardSignal & pre,
        InitialNodetoForkNode_0, reset);
    ActivityFinal: ActivityFinalNode(controlFlowOut.backwardSignal,
        JoinNodetoActivityFinal_4, reset);
    JoinNode: Join(CallDeliveringCompanyProcesstoJoinNode_3,
        CallCustomerCompanyProcesstoJoinNode_5, JoinNodetoActivityFinal_4, reset)
        ;
    ForkNode: Fork(InitialNodetoForkNode_0,
        ForkNodetoCallDeliveringCompanyProcess_1,
        ForkNodetoCallCustomerCompanyProcess_2, reset);
    CallDeliveringCompanyProcess: Delivering_Company_Process(
        ForkNodetoCallDeliveringCompanyProcess_1,
        CallDeliveringCompanyProcesstoJoinNode_3, TRUE, TRUE, container, reset);
    CallCustomerCompanyProcess: Customer_Company_Process(
        ForkNodetoCallCustomerCompanyProcess_2,
        CallCustomerCompanyProcesstoJoinNode_5, TRUE, TRUE, container, reset);

  ASSIGN
    controlFlowIn.backwardSignal := controlFlowIn.forwardSignal & InitialNode.
        ACTIVE;
    controlFlowOut.forwardSignal := ActivityFinal.FINISHED & post;

  DEFINE
```

```
    trig6 := CallDeliveringCompanyProcess.trig6;
    trig7 := CallDeliveringCompanyProcess.trig7;
    trig8 := CallDeliveringCompanyProcess.trig8;
    trig9 := CallDeliveringCompanyProcess.trig9;
    trig10 := CallDeliveringCompanyProcess.trig10;
    trig11 := CallCustomerCompanyProcess.trig11;
    trig12 := CallCustomerCompanyProcess.trig12;
    trig13 := CallCustomerCompanyProcess.trig13;
    trig14 := CallCustomerCompanyProcess.trig14;
    SUB_ACTIVE := InitialNode.ACTIVE | ActivityFinal.ACTIVE | JoinNode.ACTIVE |
        ForkNode.ACTIVE | CallDeliveringCompanyProcess.SUB_ACTIVE |
        CallCustomerCompanyProcess.SUB_ACTIVE;
    reset := resetExt | controlFlowOut.backwardSignal;
```

Listing B.2: nuXmv Module representing a “basic” *Action*

```
MODULE Action(Pre, Post, controlFlowIn, controlFlowOut, TIMEOUT)
  VAR
    state : {initial, S1, S2, S3};

  ASSIGN
    init(state) := initial;
    next(state) :=
      case
        TIMEOUT : initial;
        state = initial & controlFlowIn.forwardSignal & Pre: S1;
        state = S1 : S2;
        state = S2 & Post : S3;
        state = S3 & controlFlowOut.backwardSignal : initial;
        TRUE : state;
      esac;

    controlFlowOut.forwardSignal := (state = S3) & !TIMEOUT;
    controlFlowIn.backwardSignal := (state = initial) & controlFlowIn.
        forwardSignal & Pre & !TIMEOUT;

  DEFINE
    ACTIVE := (state != initial);
    BEH := (state = S1) & !TIMEOUT;
```

Listing B.3: nuXmv Module representing a *Send Action*

```
MODULE SendAction(Pre, Post, controlFlowIn, controlFlowOut, TIMEOUT)
  VAR
    state : {initial, S1, S2, S3};

  ASSIGN
    init(state) := initial;
    next(state) :=
      case
        TIMEOUT : initial;
        state = initial & controlFlowIn.forwardSignal & Pre: S1;
        state = S1 : S2;
        state = S2 & Post : S3;
        state = S3 & controlFlowOut.backwardSignal : initial;
        TRUE : state;
      esac;

    controlFlowOut.forwardSignal := (state = S3) & !TIMEOUT;
    controlFlowIn.backwardSignal := (state = initial) & controlFlowIn.
        forwardSignal & Pre & !TIMEOUT;

  DEFINE
    ACTIVE := (state != initial);
    BEH := (state = S1) & !TIMEOUT;
    TRIGGEROUT := (state = S1) & !TIMEOUT;
```

Listing B.4: nuXmv Module representing a *Receive Action*

```
MODULE ReceiveAction(Pre, Post, TriggerIn, controlFlowIn, controlFlowOut, TIMEOUT
    )
  VAR
    state : {initial, S1, S2, S3};
    triggerBuffer : boolean;

  ASSIGN
    init(state) := initial;
    next(state) :=
      case
        TIMEOUT : initial;
        state = initial & controlFlowIn.forwardSignal & Pre: S1;
        state = S1 & (TriggerIn | triggerBuffer) : S2;
        state = S2 & Post : S3;
        state = S3 & controlFlowOut.backwardSignal : initial;
        TRUE : state;
      esac;

    init(triggerBuffer) := FALSE;
    next(triggerBuffer) :=
      case
        state = S1 & next(state) = S2 : FALSE;
        TriggerIn : TRUE;
        TRUE: triggerBuffer;
      esac;

    controlFlowOut.forwardSignal := (state = S3) & !TIMEOUT;
    controlFlowIn.backwardSignal := (state = initial) & controlFlowIn.
        forwardSignal & Pre & !TIMEOUT;

  DEFINE
    RESET_OUT := state = S1 & (TriggerIn | triggerBuffer);
    ACTIVE := (state != initial);
    BEH := (state = S1) & (TriggerIn | triggerBuffer) & !TIMEOUT;
```

List of Figures

List of Tables

Listings

Bibliography

[1] IEEE Standard for Software and System Test Documentation. *IEEE Std 829-2008*, pages 1–150, 2008.

[2] ISO/IEC/IEEE international standard - systems and software engineering – vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, pages 1–418, 2010.

[3] IEEE standard for system and software verification and validation. *IEEE Std 1012-2012 (Revision of IEEE Std 1012-2004)*, pages 1–223, 2012.

[4] ISO/IEC/IEEE international standard - systems and software engineering – system life cycle processes. *ISO/IEC/IEEE 15288 First edition 2015-05-15*, pages 1–118, 2015.

[5] OMG Systems Modeling Language (SysML®) v2 Request For Proposal (RFP). Specification, Object Management Group (OMG), Dec. 2017.

[6] Unified modeling language (UML) version 2.5.1. Specification, Object Management Group (OMG), Dec. 2017.

[7] OMG Modeling and Analysis of Real-Time Embedded Systems (MARTE) version 1.2. Specification, Object Management Group (OMG), Apr. 2019.

[8] OMG Systems Modeling Language (SysML) version 1.6. Specification, Object Management Group (OMG), Nov. 2019.

[9] Binvox 3D mesh voxelizer, keywords: Voxelization, voxelisation, 3D model. https://www.patrickmin.com/binvox/, June 2022.

[10] Papyrus. https://www.eclipse.org/papyrus/, Apr. 2022.

[11] PRISM - Probabilistic Symbolic Model Checker. http://www.prismmodelchecker.org/, June 2022.

[12] Viewvox 3D voxel model viewer. https://www.patrickmin.com/viewvox/, June 2022.

[13] What is ACC (adaptive cruise control)? https://www.parkers.co.uk/what-is/acc-adaptive-cruise-control/, May 2022.

[14] Xtend - Modernized Java. https://www.eclipse.org/xtend/, June 2022.

[15] J.-R. Abrial. *Modeling in Event-b: System and Software Engineering*. Cambridge University Press, Cambridge, 2010.

[16] J.-R. Abrial, W. Su, and H. Zhu. Formalizing hybrid systems with event-b. In *International Conference on Abstract State Machines, Alloy, b, VDM, and z*, pages 178–193. Springer, 2012.

[17] E. Allen. “Sometimes” and “Not Never” Revisited: On Branching versus Linear Time Temporal Logic. page 28.

[18] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013.

[19] M. Askarpour, D. Mandrioli, M. Rossi, and F. Vicentini. SAFER-HRC: Safety Analysis Through Formal vERification in Human-Robot Collaboration. In A. Skavhaug, J. Guiochet, and F. Bitsch, editors, *Computer Safety, Reliability, and Security*, volume 9922, pages 283–295. Springer International Publishing, Cham, 2016.

[20] M. Askarpour, M. Rossi, and O. Tiryakiler. Co-Simulation of Human-Robot Collaboration: From Temporal Logic to 3D Simulation. *Electronic Proceedings in Theoretical Computer Science*, 319:1–8, July 2020.

[21] D. Babić and A. J. Hu. Structural Abstraction of Software Verification Conditions. In W. Damm and H. Hermanns, editors, *Computer Aided Verification*, volume 4590, pages 366–378. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[22] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, Cambridge, MA, USA, 2008.

[23] W. Bast, M. Murphree, M. Lawley, K. Duddy, M. Belaunde, C. Griffin, S. Sendall, D. Vojtisek, J. Steel, S. Helsen, L. Tratt, S. Reddy, R. Venkatesh, X. Blanc, R. Dvorak, and E. Willink. MOF Query/View/Transformation (QVT) version 1.3. Standard, Object Management Group (OMG), June 2016.

[24] C. Bocovich and J. M. Atlee. Variable-specific resolutions for feature interactions. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 553–563, New York, NY, USA, 2014. ACM.

[25] D. Borrego, R. Eshuis, M. T. Gómez-López, and R. M. Gasca. Diagnosing correctness of semantic workflow models. *Data & Knowledge Engineering*, 87:167–184, Sept. 2013.

[26] T. Bowen, F. Dworack, C. Chow, N. Griffeth, G. Herman, and Y.-J. Lin. The feature interaction problem in telecommunications systems. In *Seventh International Conference on Software Engineering for Telecommunication Switching Systems, 1989. SETSS 89.*, pages 59–62, 1989.

[27] M. Bozzano, A. C. Roberto Cavada, A. G. Michele Dorigatti, A. M. Alessandro Mariotti, M. R. Sergio Mover, and S. Tonetta. nuXmv 2.0.0 user manual. Technical report, FBK, Aug. 2021.

[28] M. Chen, P. Mishra, and D. Kalita. Coverage-driven automatic test generation for uml activity diagrams. In *Proceedings of the 18th ACM Great Lakes Symposium on VLSI - GLSVLSI '08*, page 139, Orlando, Florida, USA, 2008. ACM Press.

[29] E. Clarke, A. Fehnker, Z. Han, B. Krogh, O. Stursberg, and M. Theobald. Verification of hybrid systems based on counterexample-guided abstraction refinement. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 192–207. Springer, 2003.

[30] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In G. Goos, J. Hartmanis, J. van Leeuwen, E. A. Emerson, and A. P. Sistla, editors, *Computer Aided Verification*, volume 1855, pages 154–169. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[31] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794, 2003.

[32] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, Sept. 1994.

[33] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors. *Handbook of Model Checking*. Springer International Publishing, Cham, 2018.

[34] C. Ericson. *Real-Time Collision Detection*. Crc Press, 2004.

[35] R. Eshuis. Symbolic model checking of UML activity diagrams. *ACM Trans. Softw. Eng. Methodol.*, 15(1):1–38, Jan. 2006.

[36] R. Eshuis and P. V. Gorp. Synthesizing object life cycles from business process models. *Software and Systems Modeling*, 15(1):281–302, 2016.

[37] R. Eshuis and P. Van Gorp. Synthesizing object life cycles from business process models. *Software & Systems Modeling*, 15(1):281–302, Feb. 2016.

[38] R. Eshuis and R. Wieringa. Tool support for verifying UML activity diagrams. *IEEE Transactions on Software Engineering*, 30(7):437–447, July 2004.

[39] M. Estañol, M.-R. Sancho, and E. Teniente. Ensuring the semantic correctness of a BAUML artifact-centric BPM. 93(C):147–162, Jan. 2018.

[40] P. A. Fishwick and K. Lee. Two methods for exploiting abstraction in systems. *AI, Simulation and Planning in High Autonomous Systems*, pages 257–264, 1996.

[41] B. Foundation. Blender.org - Home of the Blender project - Free and Open 3D Creation Software. https://www.blender.org/, July 2022.

[42] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In O. Nierstrasz and M. Lemoine, editors, *Software Engineering — ESEC/FSE '99*, pages 146–162, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[43] S. Ghadai, A. Jignasu, and A. Krishnamurthy. Direct 3D printing of multi-level voxel models. *Additive Manufacturing*, 40:101929, Apr. 2021.

[44] I. Grobelna, M. Grobelny, and M. Adamski. Model checking of UML activity diagrams using a rule-based logical model. In A. Karatkevich, A. Bukowiec, M. Doligalski, and J. Tkacz, editors, *Design of Reconfigurable Logic Controllers*, pages 153–163. Springer International Publishing, Cham, 2016.

[45] J. Gutiérrez, M. Escalona, and M. Mejías. A Model-Driven approach for functional test case generation. *Journal of Systems and Software*, 109:214–228, 2015.

[46] G. Hamon, L. de Moura, and J. Rushby. Generating efficient test sets with a model checker. In *Proceedings of the Second International Conference on Software Engineering and Formal Methods, 2004. SEFM 2004.*, pages 261–270, Beijing, China, 2004. IEEE.

[47] R. Hoch, C. Luckeneder, R. Popp, and H. Kaindl. Verification of Consistency between Process Models, Object Life Cycles, and Context-dependent Semantic Specifications. *IEEE Transactions on Software Engineering*, pages 1–1, 2021.

[48] R. Hoch, M. Rathmair, H. Kaindl, and R. Popp. Verification of business processes against business rules using object life cycles. In *New Advances in Information Systems and Technologies - Volume 1 [WorldCIST'16, Recife, Pernambuco, Brazil, March 22-24, 2016].*, pages 589–598, 2016.

[49] T. P. Huck, Y. Selvaraj, C. Cronrath, C. Ledermann, M. Fabian, B. Lennartson, and T. Kröger. Hazard Analysis of Collaborative Automation Systems: A Two-layer Approach based on Supervisory Control and Simulation, Sept. 2022.

[50] ISO 26262. ISO 26262, Road vehicles – Functional safety, Nov. 2011.

[51] M. Jackson and P. Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering (TSE)*, 24(10):831–847, 1998.

[52] A. L. Juarez-Dominguez, N. A. Day, and J. J. Joyce. Modelling feature interactions in the automotive domain. In *Proceedings of the 2008 International Workshop on Models in Software Engineering*, MiSE '08, pages 45–50, New York, NY, USA, 2008. ACM.

[53] H. Kaindl, R. Hoch, M. Rathmair, and C. Luckeneder. Formal Verification of Cyber-physical Feature Coordination with Minimalist Qualitative Models. In E. Damiani, G. Spanoudakis, and L. A. Maciaszek, editors, *Evaluation of Novel Approaches to Software Engineering*, volume 1023, pages 261–287. Springer International Publishing, Cham, 2018.

[54] B. König and V. Kozioura. Counterexample-Guided Abstraction Refinement for the Analysis of Graph Transformation Systems. In H. Hermanns and J. Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920, pages 197–211. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[55] W. Lee, A. Pardo, J.-Y. Jang, G. Hachtel, and F. Somenzi. Tearing based automatic abstraction for CTL model checking. In *Computer-Aided Design, 1996. ICCAD-96. Digest of Technical Papers., 1996 IEEE/ACM International Conference On*, pages 76–81. IEEE, 1996.

[56] L. Lestingi and S. Longoni. HRC-Team: A model-driven approach to formal verification and deployment of collaborative robotic applications. 2017.

[57] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. *The Journal of Logic Programming*, 31(1):59–83, 1997.

[58] J. Li, J. Li, and F. Zhang. Model checking UML activity diagrams with SPIN. In *2009 International Conference on Computational Intelligence and Software Engineering*, pages 1–4, Dec. 2009.

[59] C. Luckeneder and H. Kaindl. Systematic top-down design of cyber-physical models with integrated validation and formal verification. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 274–275, Gothenburg Sweden, May 2018. ACM.

[60] C. Luckeneder and H. Kaindl. A case study of systematic top-down design of cyber-physical models with integrated validation and formal verification. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 1828–1836, Limassol Cyprus, Apr. 2019. ACM.

[61] P. Merson. English: Hierarchy of diagrams in UML 2.2, Apr. 2011.

[62] A. Meyer and M. Weske. Weak conformance between process models and synchronized object life cycles. In *Service-Oriented Computing - 12th International Conference, ICSOC 2014, Paris, France, November 3-6, 2014. Proceedings*, pages 359–367, 2014.

[63] F. U. Muram, H. Tran, and U. Zdun. Automated mapping of UML activity diagrams to formal specifications for supporting containment checking. *arXiv preprint arXiv:1404.0852*, 2014.

[64] J. Nellen and E. Abraham. A CEGAR approach for the reachability analysis of PLC-controlled chemical plants. In *Information Reuse and Integration (IRI), 2014 IEEE 15th International Conference On*, pages 500–507. IEEE, 2014.

[65] J. Nellen, K. Driessen, M. Neuhäußer, E. Ábrahám, and B. Wolters. Two CEGAR-based approaches for the safety verification of PLC-controlled plants. *Information Systems Frontiers*, 18(5):927–952, 2016.

[66] NuSMV. NuSMV: A new symbolic model checker manual. Technical report, NuSMV, Oct. 2018.

[67] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (Sfcs 1977)*, pages 46–57, Providence, RI, USA, Sept. 1977. IEEE.

[68] G. Preuner and M. Schrefl. Observation consistent integration of views of object life-cycles. In *British National Conference on Databases*, pages 32–48. Springer, 1998.

[69] M. Rathmair, C. Luckeneder, T. Haspl, B. Reiterer, R. Hoch, M. Hofbaur, and H. Kaindl. Formal Verification of Safety Properties of Collaborative Robotic Applications including Variability. In *2021 30th IEEE International Conference on Robot & Human Interactive Communication (RO-MAN)*, pages 1283–1288, Vancouver, BC, Canada, Aug. 2021. IEEE.

[70] M. Rathmair, C. Luckeneder, and H. Kaindl. Minimalist qualitative models for model checking cyber-physical feature coordination. In *Proceedings of the 23rd Asia-Pacific Software Engineering Conference (APSEC)*, USA, Dec. 2016. IEEE.

[71] M. Rathmair, C. Luckeneder, and H. Kaindl. Minimalist Qualitative Models for Model Checking Cyber-Physical Feature Coordination. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 233–240, Hamilton, New Zealand, 2016. IEEE.

[72] S. Rayadurgam and M. Heimdahl. Generating MC/DC adequate test sequences through model checking. In *28th Annual NASA Goddard Software Engineering Workshop, 2003. Proceedings.*, pages 91–96, Greenbelt, Maryland, USA, 2003. IEEE.

[73] K. Ryndina, J. M. Küster, and H. C. Gall. Consistency of business process models and object life cycles. In *Models in Software Engineering, Workshops and Symposia at MoDELS 2006, Genoa, Italy, October 1-6, 2006, Reports and Revised Selected Papers*, pages 80–90, 2006.

[74] D. Schramm, R. Bardini, and M. Hiller. *Vehicle Dynamics*. Springer, 2014.

[75] M. Schrefl and M. Stumptner. Behavior consistent refinement of object life cycles. In G. Goos, J. Hartmanis, J. Leeuwen, D. W. Embley, and R. C. Goldstein, editors,

Conceptual Modeling — ER '97, volume 1331, pages 155–168. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[76] J. Schumacher and M. Meyer. *Customer Relationship Management Strukturiert Dargestellt: Prozesse, Systeme, Technologien*. Springer Berlin Heidelberg, 2003.

[77] J. Seipp and M. Helmert. Counterexample-Guided Cartesian Abstraction Refinement. *Proceedings of the International Conference on Automated Planning and Scheduling*, 23(1):347–351, June 2013.

[78] J. Seipp and M. Helmert. Counterexample-Guided Cartesian Abstraction Refinement for Classical Planning. *Journal of Artificial Intelligence Research*, 62:535–577, July 2018.

[79] SPIN. SPIN Verifying Multi-threaded Software with Spin. http://spinroot.com/spin/whatispin.html, Aug. 2022.

[80] O. Stursberg, A. Fehnker, Z. Han, and B. H. Krogh. Verification of a cruise control system using counterexample-guided search. *Control Engineering Practice*, 12(10):1269–1278, 2004.

[81] C. Tian, Z. Duan, and Z. Duan. Making CEGAR more efficient in software model checking. *IEEE Transactions on Software Engineering*, 40(12):1206–1223, 2014.

[82] C. Wang, H. Kim, and A. Gupta. Hybrid CEGAR: Combining variable hiding and predicate abstraction. In *Computer-Aided Design, 2007. ICCAD 2007. IEEE/ACM International Conference On*, pages 310–317. IEEE, 2007.

[83] I. Weber, J. Hoffmann, and J. Mendling. Beyond soundness: On the verification of semantic business process models. *Distributed and Parallel Databases*, 27(3):271–343, 2010.

[84] H. Winner and M. Schopper. Adaptive cruise control. In *Handbuch Fahrerassistenzsysteme*, pages 851–891. Springer, 2015.

[85] K. Yang and X. Chen. Unsupervised learning for cuboid shape abstraction via joint segmentation from point clouds. *ACM Transactions on Graphics*, 40(4):1–11, Aug. 2021.

[86] G. Young and A. Krishnamurthy. GPU-accelerated generation and rendering of multi-level voxel representations of solid models. *Computers & Graphics*, 75:11–24, Oct. 2018.

Acronyms

CEGAR	Counter-example Guided Abstraction Refinement
CPS	Cyber-physical System
CTL	Computation Tree Logic
FI	Feature Interaction
FSM	Finite State Machine
IEEE	Institute of Electrical and Electronics Engineers
LTL	Linear Time Logic
OLC	Object Life Cycle
OMG	Object Management Group
UML	Unified Modeling Language
V&V	Verification and Validation