

# Verslag Tinlab Advanced Algorithms

**Galvin Bartes 0799967**  
176-671

10 december 2023



# Inhoudsopgave

<b>1</b>	<b>Inleiding</b>	<b>3</b>
<b>2</b>	<b>Theoretisch kader</b>	<b>4</b>
2.1	Begrippen, tools en literatuur . . . . .	4
<b>3</b>	<b>Requirements</b>	<b>7</b>
3.1	Requirements engineering . . . . .	7
3.2	Systeem omschrijving . . . . .	7
3.3	Requirements . . . . .	8
3.4	Goede requirements . . . . .	9
3.5	specificaties . . . . .	9
3.5.1	Functional vs non functional requirements . . . . .	13
3.5.2	Safety critical systems . . . . .	13
3.6	Rampen . . . . .	15
3.6.1	Therac-25 . . . . .	15
3.6.2	Ethiopian Airlines Flight 302,boeing 737 crashes . . . . .	16
3.6.3	China explosie 2015 Tianjin . . . . .	17
3.6.4	schipholbrand . . . . .	17
3.6.5	1951 . . . . .	18
3.6.6	slmramp . . . . .	18
3.6.7	Tsjernobyl . . . . .	18
3.7	De Kripke structuur . . . . .	19
3.7.1	Timed automata . . . . .	19
3.7.2	Formal testing . . . . .	20
3.7.3	Formal verification . . . . .	20
3.7.4	Formal verification environment . . . . .	21
3.7.5	Testing . . . . .	21
3.7.6	Timed automata . . . . .	22
3.7.7	CTL . . . . .	22
3.8	Guards en invarianten . . . . .	22
3.9	Deadlock . . . . .	23
3.10	Zeno gedrag . . . . .	23
<b>4</b>	<b>Logica</b>	<b>23</b>
4.1	Propositielogica . . . . .	23
4.2	Predicatenlogica . . . . .	23
4.3	Kwantoren . . . . .	24
4.4	Dualiteiten . . . . .	25
4.5	Operator: AG . . . . .	25
4.6	Operator: EG . . . . .	25
4.7	Operator: AF . . . . .	26
4.8	Operator: EF . . . . .	26
4.9	Operator: AX . . . . .	26
4.10	Operator: EX . . . . .	26
4.11	Operator: $p \cup q$ . . . . .	26

4.12	Operator: $p \ R \ q$ . . . . .	26
4.13	Fairness . . . . .	27
4.14	Safety . . . . .	27
4.15	liveness properties . . . . .	27

# 1 Inleiding

In dit verslag ga ik in op de kennis en achtergrondinformatie die nodig is voor het toepassen van Time-based model technieken. Door de toenemende complexiteit van systemen is het gebruik van modellen en de toepassing van timebased model checking op industriële controle systemen een manier van modelleren van het systeem en de requirements zodat er een bijdrage kan worden geleverd aan de acceptatie van simulatie-/modeltechniek voor de industrie.[?]. De behandelde onderwerpen zijn requirements-engineering, computational Tree Logic, propositiel logica en predicaat logica.

## 2 Theoretisch kader

In dit hoofdstuk houdt ik me bezig met de bestudering van rampen aan de hand van het vier-variabelen model maakt het analyseren mogelijk van rampsituaties. Van een aantal rampen is een beschrijving gegeven met datum, plaats en oorzaak. De analyse van de 4-variabelen modellen zal gebruikt worden voor de requirementsdefinitie, ontwerp en ontwikkeling van het sluismodel.

### 2.1 Begrippen, tools en literatuur

**Wat is uppaal** Uppaal is een geïntegreerde toolomgeving voor het modelleren, simuleren en verifiëren van real-time systemen, gezamenlijk ontwikkeld door Basic Research in Computer Science aan de Universiteit van Aalborg in Denemarken en de afdeling Informatietechnologie aan de Universiteit van Uppsala in Zweden. Het is geschikt voor systemen die kunnen worden gemodelleerd als een verzameling niet-deterministische processen met een eindige controlestructuur en klokken met reële waarde, die communiceren via kanalen of gedeelde variabelen. Typische toepassingsgebieden zijn met name real-time controllers en communicatieprotocollen, waarbij timingaspecten van cruciaal belang zijn.

UPPAAL is a verification tool for a TA based modeling language. Besides dense clocks, the tool supports both simple and complex data types like bounded integers and arrays as well as synchronization via shared variables and actions. The specification language supports safety, liveness, deadlock, and response properties.

To produce test sequences, we shall make use of UPPAAL's ability to generate diagnostic traces witnessing a submitted safety property. Currently UPPAAL supports three options for diagnostic trace generation: some trace leading to the goal state, the shortest trace with the minimum number of transitions, and fastest trace with the shortest accumulated time delay. The underlying algorithm used for finding time-optimal traces is a variation of the A-algorithm [2,12]. Hence, to improve performance it is possible to supply a heuristic function estimating the remaining cost from any state to the goal state.

**Wat is statistical model checking?** Dit verwijst naar verschillende technieken die worden gebruikt voor de monitoring van een systeem. Daarbij wordt vooral gelet op een specifieke eigenschap. Met de resultaten van de statistieken wordt de juistheid van een ontwerp beoordeeld. Statistisch model checking wordt onder andere toegepast in systeembioologie, software engineering en industriële toepassingen. [?]

**Waarom gebruiken we statistisch model checking?** Om de bovenstaande problemen te overwinnen stellen we voor om te werken met Statistical Model Checking, een aanpak die onlangs is voorgesteld als alternatief om een uitputtende verkenning van de toestandsruimte van het model te vermijden. Het kernidee van de aanpak is om een aantal simulaties van het systeem uit te voeren, deze te monitoren en vervolgens de resultaten uit het statistische gebied te gebruiken (inclusief het testen van sequentiële hypothesen of Monte Carlo-simulaties) om te beslissen of het systeem aan de eigenschap voldoet of niet. mate van vertrouwen. Van nature is SMC een compromis tussen testen en klassieke modelcontroletechnieken. Het is bekend dat op simulatie gebaseerde methoden veel minder geheugen- en tijdintensief zijn dan uitputtende methoden, en vaak de enige optie zijn. [?] Alternatieve tools voor Uppaal zijn Asynchronous Events, Vesta en MRMC.

**MODE CONFUSION** Mode confusion treedt op als geobserveerd gedrag van een technisch systeem niet past in het gedragspatroon dat de gebruiker in zijn beeldvorming heeft en ook niet met voorstellingsvermogen kan bevatten.

**Wat is automatiseringsparadox** Gemak dient de mens. Als er veel energie wordt gestoken in de ontwikkeling van hulpmiddelen die taken van werknemers overnemen heeft dat tot resultaat dat veel productieprocessen worden geautomatiseerd. De vraag is dan of vanuit mechanisch wereldpunt de robot niet de rol van de mens overneemt en of de mens nog de kwaliteiten heeft om het werk zelf te doen. [?] [?]

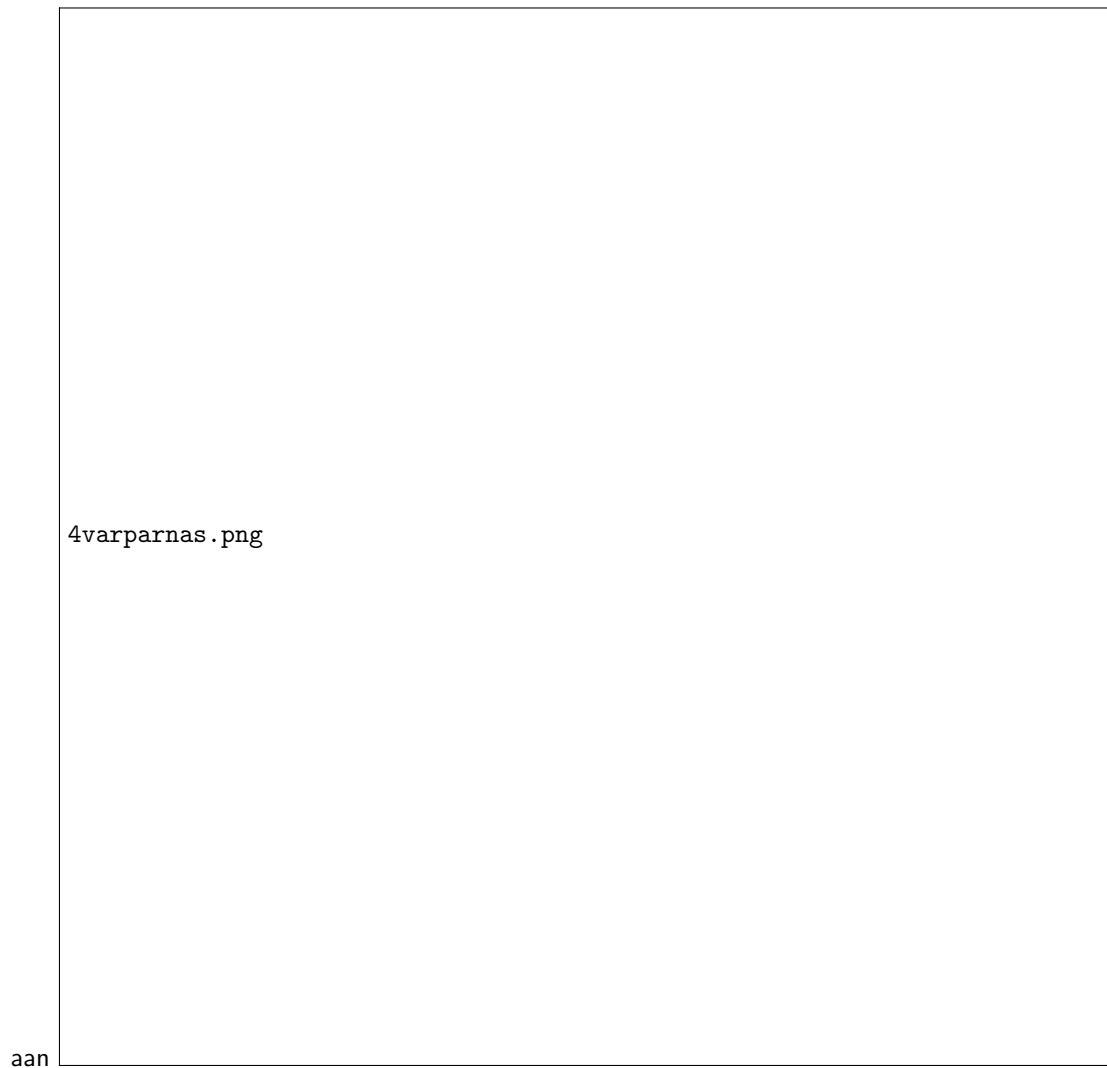
**4 variabelen model** Er zijn veel veiligheidskritische computersystemen nodig voor het bewaken en besturen van fysieke processen. Het viervariabelenmodel, dat al bijna met succes in de industrie wordt gebruikt veertig jaar, helpt het gedrag van en de grenzen tussen het fysieke te verduidelijken processen, invoer-/uitvoerapparaten en software. [?]

Het 4 variabelen model kort toegelicht Monitored variabelen: door sensoren gekwantificeerde fenomenen uit de omgeving, bijv temperatuur

Controlled variabelen: door actuatoren bestuurd fenomeen uit de omgeving Gecontroleerde variabelen kunnen bijvoorbeeld de druk en de temperatuur zijn in een kernreactor, terwijl gecontroleerde variabelen ook visuele en hoorbare alarmen kunnen zijn als het uitschakelsignaal dat een reactorsluiting initieert; wanneer de temperatuur of druk bereikt abnormale waarden, de alarmen gaan af en de uitschakelprocedure wordt gestart

Input variabelen: data die de software als input gebruikt Hier modelleert IN de ingangshardware-interface (sensoren en analoog-naar-digitaal-omzetters) en relateert waarden van bewaakte variabelen aan waarden van invoervariabelen in de software. De invoervariabelen modelleren de informatie over de omgeving die beschikbaar is voor de software. Bijvoorbeeld, IN zou een druksensor kunnen modelleren die temperatuurwaarden omzet in analoge spanningen; Deze spanningen worden vervolgens via een A/D-omzetter omgezet in gehele waarden die zijn opgeslagen in een register dat toegankelijk is voor de computer software.

Output variabelen: data die de software levert als output De uitgangshardware-interface (digitaal-naar-analoog converters en actuatoren) is gemodelleerd door OUT, dat waarden van de uitvoervariabelen van de software relateert aan waarden van gecontroleerde variabelen. Een uitvoervariabele kan bijvoorbeeld een Booleaanse variabele zijn die door de software is ingesteld met de begrip dat de waarde true aangeeft dat een reactorsluiting zou moeten plaatsvinden en de waarde false geeft het tegenovergestelde



**6 Variable model** Een uitbreiding van een 4-variabelen model is het 6-variabelen model. Optitatieve statements omschrijven de omgeving zoals we het willen zien vanwege de machine. Indicatieve statements omschrijven de omgeving zoals deze is los van de machine. Een requirement is een optitatief statement omdat ten doel heeft om de klantwens uit te drukken in een softwareontwikkel project. Domein kennis bestaat uit indicatieve uitspraken die vanuit het oogpunt van software ontwikkeling relevant zijn. Een specificatie is een optitatief statement met als doel direct implementeerbaar te zijn en ter verondersteuning van het nastreven en realiseren van de requirements. Drie verschillende type domeinkennis: domein eigenschappen, domein hypothesen, en verwachtingen. Domein eigenschappen zijn beschrijvende statementsover een omgeving en zijn feiten. Domein hypothesen zijn ook beschrijvende uitspraken over een omgeving, maar zijn aannames. Verwachtingen zijn ook aannames, maar dat zijn voorschrijvende uitspraken die behaald worden door actoren als personen, sensoren en actuators.

## 3 Requirements

### 3.1 Requirements engineering

“Requirements engineering [DoD 1991] involves all lifecycle activities devoted to identification of user requirements, analysis of the requirements to derive additional requirements, documentation of the requirements as a specification, and validation of the documented requirements against user needs, as well as processes that support these activities.” Note that requirements engineering is a domainneutral discipline; e.g., it can be used for software, hardware, and electromechanical systems. As an engineering discipline, it incorporates the use of quantitative methods, some of which will be described in later chapters of this book. Whereas requirements analysis deals with the elicitation and examination of requirements, requirements engineering deals with all phases of a project or product life cycle from innovation to obsolescence. Because of the rapid product life cycle (i.e., innovation→development→release→maintenance→obsolescence) that software has enabled, requirements engineering has further specializations for software. Thayer and Dorfman [Thayer et al. 1997], for example, define software requirements engineering as “the science and discipline concerned with establishing and documenting software requirements.”

Door middel van requirements wordt vastgelegd welke functionaliteiten het systeem moet bevatten, welke prestatieniveaus het moet bereiken, welke beperkingen moeten worden gehandhaafd en welke kwaliteitsattributen moeten worden nageleefd. Het formuleren en vaststellen van requirements is een cruciale fase in het systeemmodelleringsproces, omdat het een duidelijk beeld geeft van de gewenste systeemeigenschappen en helpt bij het bepalen van de benodigde functionaliteit en prestaties. Dit legt de basis voor verdere stappen in het modelleringsproces, zoals ontwerp, implementatie en testen.

### 3.2 Systeem omschrijving

To enable formalization (and veri

cation) of requirements, we decorate the system description with two integer variables, ErrStat and UseCase. The variable ErrStat is assigned values at unrecoverable errors: 1 if Clutch fails to close, 2 if Clutch fails to open, 3 if GearBox fails to set a gear, and 4 if GearBox fails to release a gear. The variable UseCase is assigned values whenever a recoverable error occurs in Engine: 1 if it fail to deliver zero torque, and 2 if it is not able to

nd synchronous speed. The system model is also decorated to enable veri

cation of bounded response time properties, as described in Section 3.2.

Intuitively, the

rst rule describes a local internal action transition in a component, and possibly the resetting of variables. The second rule describes synchronisation transitions that synchronise two components. The third rule describes delay transitions, i.e. when all clocks increase synchronously with time.

Reactive systems can be broadly classified as distributed systems whose subcomponents are spatially separated and concurrent systems that share resources such as processors and memories. Distributed systems communicate by message passing, whereas concurrent systems may use shared variables. Concurrent processes may share a common clock and execute in lock-step (time-synchronous systems, typical for hardware verification problems) or operate asynchronously, sharing a common processor. In the latter case, one will typically assume fairness conditions that ensure processes that could execute are eventually scheduled for execution. A common framework for the representation of these different kinds of systems is provided by the concept of transition systems. Properties of (runs of) transition systems are conveniently expressed in temporal logic.



Computerized systems pervade more and more our everyday lives. We rely on digital controllers to supervise critical functions of cars, airplanes, and industrial plants. Digital switching technology has replaced analog components in the telecommunication industry, and security protocols enable e-commerce applications and privacy. Where important investments or even human lives are at risk, quality assurance for the underlying hardware and software components becomes paramount, and this requires formal models that describe the relevant part of the systems at an adequate level of abstraction. The systems we are focussing on are assumed to maintain an ongoing interaction with their environment (e.g., the controlled system or other components of a communication network) and are therefore called reactive systems [60, 94]. Traditional models that describe computer programs as computing some result from given input values are inadequate for the description of reactive systems. Instead, the behavior of reactive systems is usually modelled by transition systems. The term model checking designates a collection of techniques for the automatic analysis of reactive systems. Subtle errors in the design of safety-critical systems that often elude conventional simulation and testing techniques can be (and have been) found in this way. Because it has been proven cost-effective and integrates well with conventional design methods, model checking is being adopted as a standard procedure for the quality assurance of reactive systems. The inputs to a model checker are a (usually finite-state) description of the system to be analysed and a number of properties, often expressed as formulas of temporal logic, that are expected to hold of the system. The model checker either confirms that the properties hold or reports that they are violated. In the latter case, it provides a counterexample: a run that violates the property. Such a run can provide valuable feedback and points to design errors. In practice, this view turns out to be somewhat idealized: quite frequently, available resources only permit to analyse a rather coarse model of the system. A positive verdict from the model checker is then of limited value because bugs may well be hidden by the simplifications that had to be applied to the model. On the other hand, counter-examples may be due to modelling artefacts and no longer correspond to actual system runs. In any case, one should keep in mind that the object of analysis is always an abstract model of the system. Standard procedures such as code reviews are necessary to ensure that the abstract model adequately reflects the behavior of the concrete system in order for the properties of interest to be established or falsified. Model checkers can be of some help in this validation task because it is possible to perform “sanity checks”, for example to ensure that certain runs are indeed possible or that the model is free of deadlocks. This paper is intended as a tutorial overview of some of the fundamental principles of model checking, based on a necessarily subjective selection of the large body of model checking literature. We begin with a case study in section 2 where the application of model checking is considered from a user’s point of view. Section 3 reviews transition systems, temporal logics, and automata-theoretic techniques that underly some approaches to model checking. Section 4 introduces basic model checking algorithms for linear-time and branching-time logics. Finally, section 5 collects some rather sketchy references to more advanced topics. Much more material can be found in other contributions to this volume and in the textbooks and survey papers [27, 28, 69, 97, 124] on the subject. The paper contains many references to the relevant literature, in the hope that this survey can also serve as an annotated bibliography.

### 3.3 Requirements

In het modelleren van systemen verwijst een requirement naar een specifieke functionaliteit, prestatie-eis, beperking of kwaliteitsattribuut dat aanwezig moet zijn in het systeem [?]. Het is een formele beschrijving van wat het systeem moet kunnen doen of aan welke specificaties het moet voldoen. Het opstellen van specificaties[?] is een belangrijk proces, omdat het de basis legt voor het ontwerp en de ontwikkeling van het systeem. Door duidelijke specificaties te hebben, kunnen de betrokkenen, zoals

ontwerpers en ontwikkelaars, een gemeenschappelijk begrip hebben van wat er van het systeem wordt verwacht. Het is ook belangrijk om de haalbaarheid van de specificaties te overwegen. Sommige eisen kunnen technisch uitdagend zijn of beperkt worden door bijvoorbeeld budgettaire of tijdsbeperkingen. Daarom moeten de specificaties realistisch en praktisch uitvoerbaar zijn.

Kortom, specificaties zijn gedetailleerde beschrijvingen van wat het systeem moet kunnen doen, welke prestaties het moet leveren en welke beperkingen er gelden. Ze dienen als een leidraad tijdens het ontwikkelingsproces en zorgen voor een duidelijke communicatie tussen de belanghebbenden over de gewenste

### 3.4 Goede requirements

Feasible A requirement is feasible if an implementation of it on the planned Valid A requirement is valid if and only if the requirement is one that the system shall (must) meet. Unambiguous A requirement is unambiguous if it has only one interpretation. Natural language tends toward ambiguity. Verifiable A requirement is verifiable if the finished product or system can be tested to ensure that it meets the requirement.

Modifiable The characteristic modifiable refers to two or more interrelated requirements or a complete requirements specification. Consistent In general, consistency is a relationship among at least two requirements. Complete A requirements specification is complete if it includes all relevant correct requirements, and sufficient information is available for the product to be built.

### 3.5 specificaties


Requirements en specificaties Assuming that the requirements are feasible with respect to the environment, Parnas and Madey [3] proposed the following acceptability condition for the software:  $NAT (NAT;SOF;OUT) REQ$  Here, the operator  $;$  is the usual composition of relations. A system implementation  $sys = IN;SOF;OUT;$  is then acceptable if and only if it satisfies the following condition:  $NAT \text{ } SYS REQ$  <https://www.sciencedirect.com/science/article/pii/S0167642315001033>

Our main aim is application of formal methods to the intelligent adaptive systems to ensure correctness of system design. 2.1 Formal methods Formal methods are mathematics based techniques for describing system properties. They provide frameworks within which people can specify, develop, and verify systems in a systematic manner. They help identify errors, ambiguities and problems in the early stages of the system development process [2]. Key elements of Formal methods 2.1.1 Formal Specification It is a complete representation of the system under study using certain formal specification languages. They provide a complete syntax, semantics and pragmatics for its representation. Different specification languages used in developing formal methods are abstract state machines, CSP, LOTOS, RAISE, Rebeca modeling language, SPARK Ada, VDM and Z notation etc.[2,3].

2.1.2 Modelling In general a model is a mathematical representation of the system under study. They can of different types such as abstract state machine models, automata based models, object oriented models etc. Further automata based models can make use of deterministic finite automata, nondeterministic finite automata, timed automata, Buchi automata,  $\omega$ -automata etc. These models are used for modelling the formally specified system. 2.1.3 Formal Verification It is a process of confirming that system under study satisfies its specifications or requirements [2]. It can be achieved by various approaches such as simulation, theorem proving and model checking. They are mainly used to check different system requirements like safety, liveness, deadlock freeness and reachability requirements.


two types of labels: atomic actions and delay actions (

4.3 Requirements In this section we list the informal requirements and desired functionality on the gear controller, provided by Mecel AB. The requirements are to ensure the correctness of the gear controller. A few operations, such as gear shifts and error detections, are crucial to the correctness. They must be guaranteed within certain time bounds. In addition, there are also requirements on the controller to ensure desired qualities of the vehicle, such as: good comfort, low fuel consumption, and low emission. 1. Performance. 2. Predictability 3. Functionality. 4. Error Detection



Schermafbeelding 2023-12-08 133303.png

Schermafbeelding 2023-12-08 132959.png



Schermafbeelding 2023-12-08 132840.png

Formeel model: Specificatie wordt gebruikt om een formele beschrijving te geven van het systeem dat wordt geanalyseerd. Het formele specificatie beschouwt een taal met een wiskundig gedefinieerde syntaxis en semantiek. In dit artikel vertegenwoordigen wij de systeemgedrag als een netwerk van getimed automaten, d.w.z eindige-toestandsmachine uitgebreid met klokvariabelen [11]. A Het systeem is gemodelleerd als een netwerk van parallel geschakelde getimed automaten. Het model is uitgebreid met begrensde discrete variabelen zijn onderdeel van de staat. Een toestand van het systeem wordt gedefinieerd door de locatie van alle automaten, de klokwaarden en de waarden van de discrete variabelen. De automaat kan een voorsprong afvuren (d.w.z. uitvoeren een overgang) afzonderlijk of synchroniseren met een andere automaat met als doel een nieuwe staat te leiden.

Hieronder specificeren we de onderscheidende elementen van de getimed automaten: Bewaker: een bepaalde uitdrukking bedoeld om een Booleaanse waarde te verifiëren expressie geëvalueerd op randen; Invariant: een toestand die de tijd aangeeft die kan zijn besteed aan een knooppunt; Kanaal: overwogen voor het synchroniseren van de voortgang van twee of meer automaten. Temporele logica:

om eigenschappen te definiëren hebben we een precieze nodig notatie.

Formele verificatieomgeving: eenmaal het model gedefinieerd en de temporele logische eigenschappen hebben we iets nodig dat dit mogelijk maakt ons om te controleren of het op tijdautomaten gebaseerde model voldoet de gedefinieerde eigenschappen. Om dit doel te bereiken overwegen we formele verificatie, een systeemproces dat gebruik maakt van wiskundig redeneren verifiëren of een systeem (dat wil zeggen het model) aan een bepaalde vereiste voldoet (dat wil zeggen, de getimede temporele eigenschappen).

### 3.5.1 Functional vs non functional requirements


Functionele requirements gaan over de functie die een systeem moet uitvoeren, terwijl non-functionele requirements betrekking hebben op de algemene eigenschappen van een systeem, zoals snelheid, bruikbaarheid, veiligheid, betrouwbaarheid. Non-functionele eisen worden ook wel systeemkwaliteiten betekenen.

### 3.5.2 Safety critical systems

De kennis van de mens is verantwoordelijk voor het scheppen van kennis. Opdoen van kennis van veiligheidsintegriteit ten overstaan van onwetendheid. Kennisdoelen zijn daarom altijd belangrijk en kunnen worden onderscheiden in 3 domeinen: cognitief, affectief en psychomotorisch.[?] Een Veiligheidsanalyse is een manier voor het evalueren van ongelukken en risico's op verschillende niveaus. Met een veiligheidsanalyse wordt gekeken naar mensen en systemen die worden blootgesteld aan een gevaar/risico en de mogelijkheid deze te minimaliseren. Veiligheid is moeilijk te definiëren omdat het een vaag concept is. Een veiligheidsanalist moet kennis hebben van het systeem, maar ook ervaring met het systeem en vooral hoe een fout in het systeem zich voordoet. Volgens Stoney[?] is veiligheid een aspect dat de veiligheid van mensenlevens of de omgeving niet in gevaar brengt. De interactie tussen systeem en omgeving levert kennis op door ad-hoc ervaring. Maar ook is kennis van abstractie en systeemdoelen belangrijk. Het kwalificeren van informatie moet ook op waarde geschat worden. Een verkeerde schatting kan fataal zijn. Zelfs als een veiligheidsanalyse is gedaan waarbij risico's zijn gedefinieerd en geprioritiseerd is een definitie van een veiligheidsniveau nodig. Soms zijn hier richtlijnen voor en soms ook niet. Er zijn voorbeelden van tools en technieken te gebruiken voor een veiligheidsanalyse. En dat zijn

1. checklists voor critical safety items
2. fault tree analysis
3. event tree analysis
4. failure modes en critical effects analysis (FMECA, FMET)
5. hazard operability studies

[?]



Schermafbeelding 2023-12-08 144723.png

Safety Critical Real-Time Systems (RTS). UML and MARTE are standardized modelling languages widely accepted by industrial designers for the design of RTS using Model-Driven Engineering (MDE). However, formal verification at early phases of the system lifecycle for UML-MARTE models remains mainly an open issue. In this paper<sup>1</sup>, we present a time properties verification framework for UML-MARTE safety critical RTS. This framework relies on a propertydriven transformation from UML architecture and behaviour models to executable and verifiable models expressed with Time Petri Nets (TPN). Meanwhile, it translates the time properties into a set of property patterns, corresponding to TPN observers. The observer-based model checking approach is then performed on the produced TPN. This verification framework can assess time properties like upper bound for loops and buffers, Best/Worst-Case Response Time, Best/Worst-Case Execution Time, Best/Worst-Case Traversal Time, schedulability, and synchronization- related properties (synchronization, coincidence, exclusion, precedence, sub-occurrence, causality). In addition, it can verify some behavioural properties like absence of deadlock or dead branches. This framework is illustrated with a representative case study. This paper

also provides experimental results and evaluates the method's performance. [?]  
[?] [?] [?] [?] [?] [?].

## 3.6 Rampen

Voor deze studie is onderzoek gedaan naar verschillende rampen aan de hand van het vier variabelen model. Elke ramp op deze manier categoriseren kan ons helpen te bepalen in hoeverre requirements een rol kunnen spelen in de veiligheid van ons model.

### 3.6.1 Therac-25

**Beschrijving** In de periode van Juni 1985 and Januari 1987 zijn er meerdere ongelukken met dodelijke afloop door de implementatie van de Therac-25 bij de behandeling van huidkanker. Dit apparaat gebruikt elektronen om stralen met hoge energie te creëren die tumoren kunnen vernietigen met minimale impact op het omliggende gezonde weefsel.

**Datum en Plaats** De ongelukken vonden plaats in de periode van juni 1985 tot en met januari 1987.

**Oorzaak** Onderzoekers constateren dat er fouten zijn gemaakt tijdens de (her-)implementatie van systemen uit eerdere productiemodellen. Terwijl de therac 20 afhankelijk was van mechanische vergrendelingen werd er bij de therac-25 software gebruikt. Onderzoekers komen daarom tot de volgende conclusies Software problemen zijn onder andere:

- slechte software engineering/designing praktijken
- de machine is afhankelijk van software voor veiligheidsoperaties
- de fout in de code is niet zo belangrijk als een geheel onveilig ontwerp
- het reinigen van de buigmagneetvariabele in plaats van aan het uiteinde van het frame
- raceconditionering om aan te geven dat het invoeren van het recept nog steeds aan de gang is
- reactie van de gebruiker
- slechte subroutines voor schermvernieuwing die foutieve informatie op de werkende console achterlieten
- Problemen met het laden van tapes bij het opstarten, waarbij het gebruik van photom-tabellen werd uitgesloten om het interlock-systeem te activeren in het geval van een laadfout in plaats van een checksum

Onderzoekers zijn van mening dat de tekortkomingen in medische apparatuur niet geheel en altijd te verwijten zijn aan softwareproblemen. Zo is er ook een rol weggelegd voor fabrikanten en overheden. Klankbordgroepen klagen over het tekort aan software-evaluaties en een tekort aan hard-copy audit trials om foutmeldingen in beeld te krijgen. [?]



### 3.6.2 Ethiopian Airlines Flight 302, Boeing 737 crashes

**Beschrijving** Ethiopische Airlines-vlucht 302 was een geplande internationale passagiersvlucht van Bole International Airport in Addis Abeba, Ethiopië naar Jomo Kenyatta International Airport in Nairobi, Kenia. Op 10 maart 2019 stortte het Boeing 737 MAX 8-vliegtuig dat de vlucht uitvoerde zes minuten na het opstijgen neer nabij de stad Bishoftu, waarbij alle 157 mensen aan boord omkwamen.

Vlucht 302 is het dodelijkste ongeval van Ethiopië Airlines tot nu toe en overtreft de fatale kaping van vlucht 961, resulterend in een crash nabij de Comoren in 1996. Het is ook het dodelijkste vliegtuigongeluk dat zich in Ethiopië heeft voorgedaan, en overtreft de crash van een Antonov An-26 van de Ethiopische luchtmacht in 1982, waarbij 73 mensen omkwamen.

Dit was het tweede MAX 8-ongeluk in minder dan vijf maanden na de crash van Lion Air-vlucht 610. Beide crashes waren aanleiding voor een twee jaar durende wereldwijde langdurige aan de grond houden van het vliegtuig en een onderzoek naar de manier waarop het vliegtuig werd goedgekeurd voor passagiersvervoer.

**Datum en Plaats** De ramp vond plaats op 10 maart 2019 nabij de stad Bishoftu

**Oorzaak** De technische oorzaak ligt bij het MCAS flight control system. Dit systeem werd geïmplementeerd om kosten te reduceren en opleidingen voor piloten in te korten. Deze single point of failure [?] werd getriggerd door een enkele angle-of-attack sensor[?]. Bij de nieuwe Boeing 737 MAX model werden tests uitgevoerd in volledige flight simulators. Nieuwe regels van het FAA instituut vereisten ondersteuning bij het uitvoeren van enkele manoeuvres. Tijdens testvluchten uitgevoerd binnen een jaar voor certificatie werd het pitch-up fenomeen geconstateerd waarop het MCAS systeem werd aangepast. In het systeem zaten nu de volgende fouten.

- MCAS wordt getriggerd door enkele sensor zonder vertraging
- Het ontwerp staat toe dat in situaties waar de angle-of-attack fout is de MCAS wordt geactiveerd
- Systeem kreeg onnodig bevoegdheid controle om de neus bij te sturen
- Waarschuwingsscherm bij fouten in de angle-of-attack werkte niet door softwarefout. Het werd ook niet kritisch bevonden door Ethiopian Airlines. Geplande updates door Boeing pas in 2020
- Een losstaande fout in de microprocessor van de controle computer kan vergelijkbare situaties doen voorkomen zonder dat MCAS wordt geactiveerd

Fouten vielen niet op omdat FAA test uitbesteedde aan Boeing. Contact tussen de organisaties FAA en Boeing verliep op management niveau. Boeing instrueerde niet alle piloten over MCAS. En het MCAS systeem werd gezien als een achtergrond systeem.

Uit onderzoek is op te maken dat problemen bij Boeing toestellen te verwijten zijn aan:

- marktdruk
- slapshot engineering
- missiekritische toepassing van technologie
- geen regelgevingsregime voor foutieve risicoanalyse

### 3.6.3 China explosie 2015 Tianjin

**Beschrijving** De explosie zorgde voor de vernietiging van 12000 voertuigen, schade aan 17000 huizen binnen een straal van 1 km. Er waren 173 doden inclusief brandweermensen. Een van de explosies zorgde voor een beving van 2.3 op de schaal van rigter. Opgeslagen materialen waren: calcium carbide, sodium nitraat, potassium nitraat, ammoniak nitraat en cyanide. Ook is er veel kritiek geweest op de acties van de autoriteiten. Zo was er censuur vanuit de overheid op de journalistiek. Ook was er naar alle waarschijnlijkheid sprake van corruptie. Zo bleek achteraf dat een van de grootste aandeelhouders Dong Shexuang de zoon te zijn van een oud-politiefchef in Tianjin haven, genaamd Dong Pijun

**Datum en Plaats** De ramp vltrok zich pp 12 augustus 2015 bij de Rulthai logistiek. Deze faciliteit zorgde voor de opslag van gevaarlijke stoffen.

**Oorzaak** De volgende factoren zouden een rol hebben gespeeld:

- Een onjuiste afbakening van het opslagmateriaal
- Er was weinig kennis bij de autoriteiten over opslagmaterialen. Zo bleek er 7000 ton aan materiaal opgeslagen, dat is ruim 70 keer te maximaal toegestane hoeveelheid.
- Onverenigbaar grondgebruik in de nabije omgeving. Veel woonwijken met naar schatting 6000000 bewoners en 500 lokale bedrijven in de buurt van de opslag gevaarlijke stoffen.

### 3.6.4 schipholbrand

**Beschrijving** Bij de schipholbrand op 27/10/2005 vielen zeker 11 doden onder migranten in de cellencomplexen van schiphol-oost. Doodsoorzaak van de slachtoffers is verstikking. Het gebouw voldeed niet aan de eisen voor brandveiligheid, personeel was niet goed getraind voor dergelijke situaties en de hulpverlening kwam door verschillende factoren te laat op gang.

**Datum en Plaats** De ramp voltrok zich bij de vleugels j en k van het cellencomplexen van schiphol-oost op 27 oktober 2005 .

**Oorzaak** Doordat de deur van cel 11 niet werd gesloten kreeg de brand zuurstof. De brand kon zich uitbreiden in de aanwezigheid van grote hoeveelheid brandbare materialen. Deze fungeerde als brandstof waardoor de brand zich verder kon ontwikkelen zowel binnen als buiten de cel. De dakluiken werkten met als gevolg dat de rook en warmte niet werden afgevoerd. Dit heeft het mede onmogelijk gemaakt alle celdeuren te openen en daardoor de reddingsoperatie van de bewaarders belemmerd. Door de schilconstructie van het cellencomplex was het mogelijk dat de brand zich kon uitbreiden naar de andere cellen en naar de gang.

Dit is onopgemerkt gebleven doordat er geen specifieke eisen werden gesteld bij de bouw van vleugels j en k. Was er te weinig aandacht voor bouwregels en onderzoek naar eerdere branden in het cellencomplex. Bovendien waren bHV'ers niet goed opgeleid en was er geen goede afstemming met de brandweer. Er was geen evacuatieplan dat dat rekening hield met de overplaatsing van gedetineerden naar andere penitentiaire inrichtingen. En had men te weinig aandacht voor de traceerbaarheid van celbewoners en bovendien voor ademhalingsproblemen van gedetineerden.

### 3.6.5 1951

**Beschrijving** Turkish Airlines-vlucht 1951 (ook bekend als TK 1951) was een passagiersvlucht die op woensdag 25 februari 2009 neerstortte in de buurt van het Nederlandse vliegveld Schiphol. Hierbij kwamen 9 van de 135 inzittenden om het leven.

Het vliegtuig, een Boeing 737-800 van de Turkse maatschappij Turkish Airlines, was op weg van het vliegveld Istanbul Atatürk naar Schiphol, maar stortte om 10:26 uur, kort voor de landing op de Polderbaan, neer op een akker en brak daarbij in drie stukken.

**Datum en Plaats** Op woensdag 25 februari 2009 voltrok zich de ramp met een toestel van turkisch airlines tijdens vlucht 1951.

**Oorzaak** De automatische reactie van het toestel werd getriggerd door een fout gevoelige radio altimeter waardoor de automatische gashendel de energiemotor op actief stelde. Inadequaet handelen van de piloten ondanks een defecte hoogtemeter en onvolledige instructies van de luchtverkeersleiding.

Het officiële onderzoek van de Onderzoeksraad voor Veiligheid wees inadequaet handelen van de piloten als hoofdoorzaak aan voor het ongeluk. Ondanks een defecte hoogtemeter en onvolledige instructies van de luchtverkeersleiding hadden de piloten het ongeluk kunnen voorkomen, aldus de Onderzoeksraad.

### 3.6.6 slm ramp

**Beschrijving** Toen het toestel op 07/06/1989 de Anthony Nesty Zanderij naderde, was het daar, anders dan het weerbericht had voorspeld, mistig. Het zicht was evenwel niet zo slecht dat er niet op zicht kon worden geland. Gezagvoerder Will Rogers besloot echter via het Instrument Landing System (ILS) te landen, hoewel dit niet betrouwbaar was en hij voor zo'n landing ook geen toestemming had. De gezagvoerder brak drie landingspogingen af. Bij de vierde poging negeerde de bemanning de automatische waarschuwing (GPWS) dat het toestel te laag vloog. Het toestel raakte op 25 meter hoogte twee bomen. Het rolde om de lengte-as en stortte om 04.27 uur plaatselijke tijd ondersteboven neer.

**Datum en Plaats** De ramp voltrok zich op 07 juni 1989 toen het vliegtuig de Anthony Nesty Zanderij naderde

### Oorzaak

Uit onderzoek bleek dat de papieren van de bemanning niet in orde waren door nalatigheid in de crew-member screening bij de SLM. Geconcludeerd werd dat de gezagvoerder roekeloos had gehandeld door voor een ILS-landing te kiezen terwijl hij daar geen toestemming voor had, en door onvoldoende op de vlieghoogte te hebben gelet. Het vliegen onder de minimum hoogte leidde tot collision met een boom.

### 3.6.7 Tsjernobyl

**Beschrijving** De mislukte veiligheidscontrole die 26 april 1986 01.24 uur in de Sovjet-Unie leidde tot explosies in een van de reactoren in de kerncentrale. De reactoren hadden geen veiligheidsomhulling en de reactor bevat grote hoeveelheden brandbaar grafiet. Door de explosie en de brand kwamen er radioactieve stoffen vrij. Het gaat helemaal mis in de kernreactor 4. De warmteproductie nam toe met een explosie tot gevolg. 31 mensen kwamen om, waaronder veel mensen dagen later door stralingsziekte.

**Datum en Plaats** De ramp van Tjernobyl voltrok zich op 26 april 1986 [?].

### Oorzaak

Technici bij kerncentrale 4 voerden een slecht opgezet/ ontworpen experiment uit. De kracht regulering werd uitgeschakeld evenals veiligheidssystemen. Door een bedieningsfout in een testprocedure werd het vermogen van de koelinstallaties negatief beïnvloed. Mede door een ontwerpfout in de nood-stopprocedure kon in het systeem niet snel genoeg schakelen om remmende invloed uit te oefenen op het toenemende vermogen van de reactorkernen. Met brand en explosies tot gevolg. [?] [?] [?] [?] [?]

Modellen

## 3.7 De Kripke structuur

De kripke structuur is een set van locaties, transities, guards, klokken en data-variabelen. Temporal logic formele taal voor het specificeren en redeneren over hoe de Het gedrag van een systeem verandert in de loop van de tijd breidt de propositiologica uit met modale/temporele operatoren een belangrijk gebruik: representatie van toekomstige systeemeigenschappen gecontroleerd door een modelchecker. Informeel betekenen A en E 'langs alle paden' en 'langs ten minste één pad', respectievelijk; terwijl X, F, G en U verwijst naar 'volgende staat', 'een toekomstige staat', 'hele toekomst' staten," en "Tot." We gebruiken structurele inductie op CTL-formules om te definiëren een tevredenheidsrelatie:  $M, s$  impliceert  $\phi$  dat hangt af van een transitiesysteem  $M = (S, \rightarrow, L)$ , a toestand  $s$  van  $M$ , en een CTL-formule  $\phi$ . Syntaxis van CTL is opgesplitst in state- en padformules specificeer respectievelijk eigenschappen van toestanden/paden een CTL-formule is een toestandsformule State formulae: waarbij een E AP en een padformule is voor een atomaire propositie Path formulae waarbij een toestandsformule (stateformula) is Elke padkwantificator moet worden gevolgd door een temporele kwantificator in de syntactische boom van elke formule

### 3.7.1 Timed automata

A timed automaton is a (non-deterministic) finite state automaton composed of edges and vertexes and extended with real-valued clocks and integer variables. Clocks proceed at the same rate and measure the time since they were reset. Integer variables are finite domain variables with rate zero.

Automata learning algorithms have received significant attention from the verification community in the past two decades. Besides being used to improve the efficiency and scalability of compositional verification, automata learning has also been successfully applied in other aspects of verification: among others, it has been used to automatically generate interface models of computer programs [7], to learn a model of the traces of the system errors for diagnosis purposes [27], to find bugs in the implementation of network protocols [39], to extract behavior model of programs for statistical program analysis [29], and to do model-based testing and verification [81, 107]. Later in 2017, Frits Vaandrager [102] explains the concept of model learning used in the above applications. In order to be of practical use, the learning algorithms have to be computationally efficient and easily adaptable to the different learning scenarios. On the one hand, with more complex tasks at hand, some researchers have proposed several optimizations to improve the efficiency of finite automata learning, such as learning algorithms based on classical

cation trees [59,63], efficient counterexample analysis for learning algorithms [87], learning algorithm NL for nondeterministic finite automata (NFAs) [20], and learning algorithms for alternating finite automata [10].

### 3.7.2 Formal testing

Index Terms—SCADA, model checking, formal methods, timed automaton, temporal logic, critical infrastructure, security, safety We have worked on projects for a broad range of application domains; e.g., medical equipment, factory automation, transportation, communications, automotive. documented, traced, reviewed, and tested

### 3.7.3 Formal verification

In order to support the formal verification based on MC, a timed automata should be created to express the system behavior, supporting the evaluation of different properties such as safety, reachability, liveness, and deadlock [10]. However, generating these representations is not a simple task and requires sufficient knowledge of the design team to correctly express the system properties. To automate the timed automata construction, it is our claim that a model transformation can be performed using as input the architectural representation.

An implementation, IMP, consists of a description of the actual hardware design that is to be verified. A specification, SPEC, is a description of the intended/required (properties) behavior of a hardware design. Formal Verification (FV) is a systematic process of ensuring, through exhaustive algorithmic techniques, that a design implementation IMP satisfies the properties established in its specification SPEC. With formal verification, we are able to overcome both of simulation's challenges: the observability and the controllability, since formal verification algorithmically and exhaustively explores all possible input values over time. The simulation is empirical - meaning you use trial and error to discover bugs. It would take an intractable amount of time to test all possible combinations. Therefore, it is never complete. However the verification engineers are focusing their effort on how to break the design, not on what the design is intended to do. Formal verification, on the other hand, is mathematical and exhaustive and allows the engineer to focus solely on intent, or "what is the design's correct behavior". The main disadvantage of the formal verification is its limited capacity when the explosion state [6] is present. There are two types of formal verification: • EQUIVALENCE CHECKING: It is concerned with the verification that two representations of the same design are equivalent. Typically, it is gate level versus another gate level or RTL • MODEL CHECKING: Specification is in the form of a temporal logic formula, the truth of which is determined with respect to a semantic model provided by an implementation.

The main validation methods for complex systems are a simulation, testing, deductive verification, and model checking. Simulation and testing [82] both involve making experiments before using the system in the field. While simulation is carried out on an abstraction or a model of the system, testing is done on the actual product. In the case of circuits, simulation is carried out on the design of the circuit, whereas testing is done on the circuit itself. In both situations, these techniques inject signals at specific points in the system and observe the resulting signals at other points. For software, simulation and testing generally involve preparing determined inputs and observing the corresponding outputs. These techniques can be a cost-efficient method to find many errors. However, checking all of the possible interactions and potential pitfalls utilizing simulation and testing techniques is rarely possible. This problem of design validation (ensuring the correctness of the design at the earliest stage possible) is the main challenge in any responsible system development process, and the tasks intended for its solution occupy a huge portions of the development cycle cost and time budgets. The currently practiced techniques for design

validation in most places are still the veteran methods of simulation and testing. Although provably effective in the very early stages of debugging, when the design is still infested with multiple bugs, their effectiveness drops quickly as the design becomes cleaner, and they require an alarmingly increasing amount of time to uncover the more subtle bugs. A principal disadvantage of these methods is that one can never be sure when a limit has been reached or even an estimate of how many bugs may still lurk in the design. As the complexity of designs increases tremendously, say from 0.5 to 5 million gates per chip, some far-seeing managers predict the complete collapse of these conventional methods and their total inability to scale up [53]. The approach of formal verification is a very attractive and dramatically appealing alternative to simulation and testing. While simulation and testing explore some of the possible behaviors and scenarios of the system, the question of whether the unexplored trajectories contain the fatal bug remains open. Formal verification on the other hand conducts an exhaustive exploration of all possible behaviors. Thus, when a design is pronounced correct by a formal verification method, it means that all the possible behaviors have been explored, so that the questions of proper coverage or a missed behavior become irrelevant. One of the main approaches to formal verification is model checking. In model checking a desired behavioral property is verified over a given system (the model) through exhaustive (explicit or implicit) enumeration of all the states reachable by the system and the behaviors that lead to them. Compared to other approaches, model checking has three important advantages. First, it is fully automatic, and its application requires no user supervision or expertise in mathematical disciplines such as logic and theorem proving. Anyone who can run simulations of a design is fully eligible and capable of model-checking the same design. In the context of the currently practiced method, model checking can be categorized as the ultimately superior simulation tool. Secondly, when the design fails to satisfy the desired property, the process of model checking always generates a counterexample that shows a behavior which falsifies the property. This faulty trace offers a priceless insight into understanding the real reason for the failure as well as significant clues for solving the problem. Last but not least, model checking has a sound and mathematical underpinning. These important advantages together with the advent of symbolic model checking, which allows the exhaustive implicit enumeration of an astronomic number of states, completely revolutionized the field of formal verification and transformed it from a purely academic discipline into a viable practical method that can potentially be integrated as an vital technique for design validation in a lot of industrial development processes [53].

#### 3.7.4 Formal verification environment

Formal Verification Environment: once defined the model and the temporal logic properties, we need something enabling us to check whether the timed automata based model satisfies the defined properties. To this aim we consider formal verification, a system process exploiting mathematical reasoning to verify if a system (i.e., the model) satisfies some requirement (i.e., the timed temporal properties).

#### 3.7.5 Testing

Testing is the execution of the system under test in a controlled environment following a prescribed procedure with the goal of measuring one or more quality characteristics of a product, such as functionality or performance. Testing is the primary software validation technique used by industry today. However, despite the importance and the many resources and man-hours invested by industry (about 30% effort), testing remains quite ad hoc and error prone. We focus on conformance testing i.e., checking by means of execution whether the behavior of some black box implementation conforms to that of its specification, and moreover doing this within minimum time. A promising approach to improving the effectiveness of testing is to base test generation on an abstract formal model of the system under test

(SUT) and use a test generation tool to (automatically or user guided) generate and execute test cases. Model based test generation has been under scientific study for some time, and practically applicable test tools are emerging [4,16,18,10]. However, little is still known in the context of real-time systems.

### 3.7.6 Timed automata

### 3.7.7 CTL

CTL, geïntroduceerd door Emerson en Clarke [18], is een tijdelijke vertakkingstijd logica die wordt gebruikt als specificatietaal voor eindige-toestandssystemen. De executies van het systeem worden gemodelleerd als lineaire reeksen systeemgebeurtenissen. Die gebeurtenis reeksen worden in de onderliggende berekeningsboom rekenpaden genoemd het modelleren van de structuur van de tijd. CTL-formules zijn samengesteld uit logische operatoren, padkwantificatoren en tijdelijke operatoren. De padkwantificatoren worden gebruikt om aan te geven of een eigenschap dat zou moeten doen houdt een bepaald rekenpad (E) of alle rekenpaden (A) vast uit de huidige staat. De temporele operatoren worden gebruikt om eigenschappen te beschrijven van een pad door de berekeningsboom.

Computatieboomlogica (CTL) is een vertakkingstijdlogica dat omvat zowel de propositionele connectieven als temporele verbindingen AX, EX, AU, EU, AG, EG, AF, en EF. We moeten aantonen dat een real-time programma voldoet aan de eisen opgesteld en gespecificeerd.

Een formele verificatie van de gespecificeerde requirements ofwel modeleigenschappen wordt gerealiseerd door deze te vertalen naar de query-language van de symbolic model-checker Uppaal. [?],[?].

Temporele logica is een al lang gebruikt en goed begrepen concept om softwarespecificaties en computerprogramma's te modelleren door middel van toestandsvergangsemantiek. Een pad is een reeks toestanden. Bij elke toestand kan een toestandsformule worden geëvalueerd. Een vertakkende tijdpadformule strekt zich uit over mogelijke reeksen van opeenvolgende toestanden. States hebben labels. Stateformule worden opgebouwd uit deze atomaire formules met behulp van de Booleaans e kwantificatoren, met de beperking dat we een padlogische kwantificator in een stateformula niet kunnen ontkennen.

Het bewijs van de modeleigenschappen kan wiskundig worden opgesteld met behulp van kwantoren, zoals:

## 3.8 Guards en invarianten

Invarianten en guards zijn condities. Een invariant is een conditie die geldt in een state en die altijd waar is wanneer de automaat zich in die state bevindt. Een guard is een conditie die geldt in een transitie. M.a.w. De transitie kan alleen genomen worden wanneer de guard geldig is.

In feite zijn veiligheidseigenschappen een soort invarianten. Invarianten zijn eigenschappen die worden gegeven door een voorwaarde voor de toestanden en vereisen dat geldt voor alle bereikbare staten. Het begrip 'invariant' kan dus als volgt worden uitgelegd: er moet aan de voorwaarde zijn voldaan door alle begintoestanden en de bevrediging van is onveranderlijk onder alle overgangen in het bereikbare pad van het gegeven transitiesysteem.

Invarianten kunnen worden gezien als toestandseigenschappen en kan worden gecontroleerd door te kijken naar de bereikbare staten.

### 3.9 Deadlock

Deadlock is een bereikbare staat waarin het systeem helemaal geen acties kan uitvoeren – Een Deadlock hangt af van de reeks acties die een bereikbare staat niet kan uitvoeren. Om de impasse te behouden moet A niet alleen overschatten wat P kan doen, maar ook wat P weigert

### 3.10 Zeno gedrag

Zeno gedrag houdt in dat de mogelijkheid dat in een eindige hoeveelheid tijd een oneindig aantal handelingen kan worden verricht.

## 4 Logica

### 4.1 Propositielogica

In de propositielogica onderzoekt men het waarheidsgehalte van samengestelde uitspraken aan de hand van elementaire proposities en logische voegwoorden. Wij noemen de woorden 'en', 'of', 'als . . . dan . . .', 'impliceert' logische voegwoorden. Hoewel het taalkundig geen voegwoorden zijn, noemen wij de ontkenningen 'niet' en 'geen' toch logische voegwoorden.

In de propositielogica worden proposities gemaakt van elementaire proposities en logische voegwoorden. In de symbolische propositielogica, de propositierekening, wordt een propositieformule gemaakt van variabelen (letters), logische operatoren en haakjes. Net zoals een formule in de rekenkunde, heeft het deel van een propositieformule binnen de haakjes een hogere prioriteit dan het deel buiten de haakjes. Bovendien wordt alles wat tussen haakjes staat, beschouwd als een eenheid. Om een propositieformule correct samen te stellen, moeten wij verplicht gebruik maken van de volgende grammaticale regels:

1. Een variabele is een formule;
2. Als  $p$  een formule is, dan is  $\neg p$  een formule;
3. Als  $p$  en  $q$  formules zijn, dan zijn  $(p \wedge q)$ ,  $(p \vee q)$ ,  $(p \rightarrow q)$  en  $(p \leftrightarrow q)$  formules.

In een propositieformule mogen haakjes worden weggelaten mits er geen verwarring ontstaat

Tijdens het logisch beschrijven van logica of bij het maken van formules over formules, is het mogelijk dat wij een paradoxale uitspraak doen. Zo'n uitspraak is vergelijkbaar met de volgende zin: Deze zin is onwaar. De bovenstaande zin beschrijft zichzelf. Is deze zin nu waar of onwaar? Dit is een paradox. Wij moeten de uitspraken over de logica scheiden van de logica zelf. Daarom introduceren wij metasymbolen. Deze metasymbolen zijn geen onderdeel van de beschreven logica, maar een toevoeging aan de omgangstaal in dit dictaat.

Voor het bepalen van de waarheidswaarde van een formule  $f(p_1; p_2; \dots; p_n)$  moeten alle mogelijke combinaties van waardetoekenningen worden bepaald. Deze combinatie van waardetoekenningen vormen samen de waarheidstabel van deze formule.

### 4.2 Predicatenlogica

Hoewel de volgende redenering in de propositielogica ongeldig is, wordt zij intuïtief toch als geldig beschouwd:



1.  $f_1$  "alle republieken plegen geen overspel"
2.  $f_2$  "sommige overspeligen zijn president"
3.  $y$  "sommige presidenten zijn geen republiek"

De geldigheid van deze redenering is gebaseerd op informatie, waarmee de propositielogica geen rekening mee houdt. Om deze extra informatie bij het redeneren te betrekken, moeten wij de propositielogica uitbreiden met de begrippen eigenschappen, variabelen en kwantoren. De zin "alle mensen zijn sterfelijk" geeft aan dat objecten, die de eigenschap 'menselijk zijn' hebben, blijkbaar ook de eigenschap 'sterfelijk zijn' hebben. uitspraak symbolisch:

1. sterfelijk objecten  $S(x)$
2. menselijke objecten  $M(x)$
3.  $x$  is een priemgetal  $P(x)$

In de uitspraken geven wij de eigenschappen sterfelijk en priemgetal aan met de hoofdletters  $S$  en  $P$ . De variabele  $x$  stelt objecten voor. Een variabele, waarvan de waarde onbekend is, noemen wij vrije variabele. Definitie 4.1 (Predikaat) Een predikaat is een uitspraak met vrije variabelen, die een propositie wordt zodra alle vrije variabelen in dat predikaat gebonden zijn aan een waarde.

### 4.3 Kwantoren

Kwantificator wordt gebruikt om de variabele van predikaten te kwantificeren. Het bevat een formule, een soort verklaring waarvan de waarheidswaarde kan afhangen van de waarden van sommige variabelen. Wanneer we een vaste waarde aan een predikaat toekennen, wordt het een propositie. Op een andere manier kunnen we zeggen dat als we het predikaat kwantificeren, het predikaat een propositie wordt. Kwantificeren is dus een woordsoort dat verwijst naar kwantificeringen als 'alles' of 'sommige'.

Er zijn hoofdzakelijk twee soorten kwantoren: universele kwantoren en existentiële kwantoren. Daarnaast hebben we ook andere soorten kwantoren, zoals geneste kwantoren en kwantoren in standaard Engels gebruik. Kwantificator wordt voornamelijk gebruikt om aan te tonen dat voor hoeveel elementen een beschreven predikaat waar is. Het laat ook zien dat voor alle mogelijke waarden of voor sommige waarde(n) in het universum van het discours het predikaat waar is of niet.

Tot dusverre hebben wij in de voorbeelduitspraken "voor alle  $x$ " en "er is een  $x$ " gekoppeld aan alle voorkomens van  $x$  in een universum. De uitspraak "er is een  $x$  in het universum van  $x$  met de eigenschap  $P$ " wordt weergegeven als:

Soms willen wij zo'n universum beperken. Zo'n beperkt deel van een universum, een domein, is het gebied waaruit de gebonden variabele moet putten. De beschrijving van het domein  $D(x)$  wordt onder de kwantor geplaatst.

Met de universele kwantor  $\forall x$  bedoelen wij alle waarden van  $x$  binnen het opgegeven domein. Als het domein niet wordt opgegeven, dan bedoelen wij een of meer waarden  $x$  uit een eindig universum. Bijvoorbeeld:

Met de existentiële kwantor  $\exists x$  bedoelen wij 'e' en of meer waarden van  $x$  uit het opgegeven domein. Als het domein niet wordt opgegeven, dan bedoelen wij een of meer waarden  $x$  uit een eindig universum. Bijvoorbeeld:

Een overzicht van meerde temporal operators:

1. F sometime in the Future, The future time operator (F) is used to specify that some property eventually holds at some state in our path
2. G Globally in the future, The global operator (G) is used to specify that some property holds for all states of a particular path.
3. X neXtime, The next time operator (X) is used to specify that some property holds in the second state of a path.
4. U until, The until operator (U) is a binary operator, and is used to specify that the first property holds in all states preceding the one where the second property is satisfied
5. R the release operator, The release operator (R) is also a binary operator, and is used to specify that the second property holds in all states along a path up to and including the first state that satisfies the first property. The first property is not required to be eventually satisfied.

Path quantifiers:

1. E there Exists a path
2. A in All paths

#### 4.4 Dualiteiten

Het dual van elke uitspraak in een Booleaanse algebra B is de uitspraak die wordt verkregen door de bewerkingen uit te wisselen  $+$  en  $,$  en hun identiteitselementen 0 en 1 in de oorspronkelijke verklaring verwisselen. De dubbele van bijvoorbeeld  $(1 + a) (b + 0) = b$  is  $(0 a) + (b 1) = b$ . Observeer de symmetrie in de axioma's van een Booleaanse algebra B. Dat wil zeggen, de duale van de reeks axioma's van B is de hetzelfde als de originele set axioma's. Dienovereenkomstig geldt het belangrijke principe van dualiteit in B. Namelijk: Stelling 15.1 (Dualiteitsprincipe): De dualiteit van elke stelling in een Booleaanse algebra is ook een stelling. Met andere woorden: als een uitspraak een gevolg is van de axioma's van een Booleaanse algebra, dan is de duale ook een gevolg van deze axioma's, aangezien de dubbele verklaring kan worden bewezen door de duale van elke stap van het bewijs te gebruiken van de oorspronkelijke verklaring.

#### 4.5 Operator: AG

AG P: in alle mogelijke paden, is P altijd waar

#### 4.6 Operator: EG

EGp - in tenminste een pad geldt p is overal geldig. Een pad is een reeks toestanden. Bij elke toestand kan een toestandsformule worden geëvalueerd. Een vertakkende tijdpadformule strekt zich uit over een enkele reeks van opeenvolgende toestanden. In de huidige reeks heeft elke toestand een label p, en is de atomaire toestandsformule p waar in die toestand behorend tot deze reeks.

## 4.7 Operator: AF

AF  $p$  - voor alle paden geldt,  $p$  is ergens in de toekomst/ ooit geldig.

Een pad is een reeks toestanden. Bij elke toestand kan een toestandsformule worden geëvalueerd. Deze vertakkende tijdpadformule strekt zich uit over meerdere paden met in de toekomst een geldige opvolgende toestand. Als voor alle paden geldt dat deze vanaf de huidige toestand er in de toekomst een toestand een label  $p$  heeft, dan is de atomaire toestandsformule  $p$  waar in die toestand.

## 4.8 Operator: EF

EF, er is een pad waar uiteindelijk, vanaf de huidige staat met enkele andere staten  $p$  geldig is EF  $P$ : in tenminste 1 pas, zal  $P$  vroeg of laat geldig zijn

We hebben hier een pad met een reeks toestanden. Waarbij ooit op dit pad een toestand kan toestandsformule worden geëvalueerd welke een label  $p$  heeft, en de atomaire toestandsformule  $p$  waar is in die toestand.

## 4.9 Operator: AX

AX $p$  - voor alle paden is  $p$  in de volgende state geldig

We spreken hier van alle mogelijke paden waarvoor geldt dat in de volgende toestand kan een toestandsformule worden als "waar" wordt geëvalueerd. States hebben labels. Als de volgende toestand een label  $p$  heeft, dan is de atomaire toestandsformule  $p$  waar in die toestand.

## 4.10 Operator: EX

EX  $P$ : in ten minste 1 execution path, is  $p$  geldig in de volgende state Een pad is een reeks toestanden. Bij elke toestand kan een toestandsformule worden geëvalueerd. Een vertakkende tijdpadformule strekt zich uit over een mogelijke reeks van opeenvolgende toestanden. Waarbij geldt dat in de huidige reeks er een toestand is met een opvolgende state met een label  $p$  heeft, en is de atomaire toestandsformule  $p$  waar in die toestand.

## 4.11 Operator: $p \text{ U } q$

$A[P \text{ U } Q]$ : in alle uitvoerpaden is  $P$  geldig totdat  $Q$  geldig is. Of  $E[P \text{ U } Q]$ : in tenminste 1 pad is  $P$  geldig totdat  $Q$  geldig is. Een pad is een reeks toestanden. Bij elke toestand kan een toestandsformule worden geëvalueerd. Een vertakkende tijdpadformule strekt zich in geval van  $A[P \text{ U } Q]$ : uit over alle reeksen van opeenvolgende toestanden. Waarvoor geldt dat als de toestand een label  $p$  heeft, dan is de atomaire toestandsformule  $p$  waar in die toestand tot aan de toestand waarin een toestand het label  $q$  heeft.

## 4.12 Operator: $p \text{ R } q$

$R$  ("release") is de logische duaal van de  $U$ -operator. De operator vereist dat het tweede argument stand moet houden tot en met de eerste staat waar het eerste argument geldt. Het eerste argument is dat niet nodig om uiteindelijk waar te worden.

Een pad is een reeks toestanden. Bij elke toestand kan een toestandsformule worden geëvalueerd. Een vertakkende tijdpadformule strekt zich uit over een mogelijke reeks van opeenvolgende toestanden.

States hebben labels. Als de huidige toestand een label  $q$  heeft, dan is de atomaire toestandsformule  $q$  waar in die toestand tot aan de toestand waarin  $p$  geldig is. Met de voorwaarde dat de eigenschap  $p$  niet gelabeld hoeft de worden aan een opvolgende state. De release-operator ( $R$ ) is ook een binaire operator en betekent dat de tweede argument geldig moet zijn in alle staten tot aan  $q$  inclusief de eerste staat die voldoet aan de eerste eigenschap.

#### 4.13 Fairness

Deze eigenschappen beschrijven de vereiste dat een proces vooruitgang boekt in de richting van een specifiek doel, waarvan de verwezenlijking afhangt van de fairness van het systeem. Als een proces nooit wordt uitgevoerd, kan het zijn doel meestal niet bereiken. Daarom worden deze eigenschappen vaak alleen langs eerlijke paden geëvalueerd, d.w.z. paden die voldoen aan een van de volgende drie vereisten voor fairnessconstraints: Absolute Fairness, Impartiality: every process should be executed infinitely often: Strong Fairness: every process that is infinitely often enabled should be executed infinitely often in a state where it is enabled: Weak Fairness: every process that is almost always enabled should be executed infinitely often:

Recall that fairness assumptions (see Section 3.5) are used to rule out certain computations that are considered to be unrealistic for the system under consideration. These unreasonable computations that ignore certain transition alternatives forever are the “unfair” ones; the remaining computations are thus the “fair” ones. As there are different notions of fairness, various distinct forms of fairness can be imposed: unconditional, strong, and weak fairness constraints.

#### 4.14 Safety


Safety properties are often characterized as “nothing bad should happen”. The mutual exclusion property—always at most one process is in its critical section—is a typical safety property. It states that the bad thing (having two or more processes in their critical section simultaneously) never occurs. Another typical safety property is deadlock freedom. For the dining philosophers (see Example 3.2, page 90), for example, such deadlock could be characterized as the situation in which all philosophers are waiting to pick up the second chopstick. This bad (i.e., unwanted) situation should never occur.

#### 4.15 liveness properties


Veiligheidseigenschappen geven aan wat er wel of niet mag voorkomen, maar eis niet dat er ooit iets gebeurt. liveness geeft aan wat er moet gebeuren. De eenvoudigste vorm van een liveness-eigenschap garandeert dat er uiteindelijk iets goeds gebeurt. Met andere woorden: er bestaat een tijdstip waarop het systeem zich in de

Schermafbeelding 2023-12-08 132840.png

Schermafbeelding 2023-12-08 132959.png



Schermafbeelding 2023-12-08 133123.png



Schermafbeelding 2023-12-08 133201.png




Schermafbeelding 2023-12-08 133242.png

Schermafbeelding 2023-12-08 142543.png


Schermafbeelding 2023-12-08 142602.png

Schermafbeelding 2023-12-08 142800.png



Schermafbeelding 2023-12-08 144259.png

Schermafbeelding 2023-12-08 144315.png




Schermafbeelding 2023-12-08 144343.png


Schermafbeelding 2023-12-08 144403.png




Schermafbeelding 2023-12-08 144418.png




Schermafbeelding 2023-12-08 144433.png



Schermafbeelding 2023-12-08 144451.png




Schermafbeelding 2023-12-08 144519.png



Schermafbeelding 2023-12-08 144537.png

Schermafbeelding 2023-12-08 144614.png



Schermafbeelding 2023-12-08 144758.png

Schermafbeelding 2023-12-08 144815.png