Contract Machines: An Engineer-friendly Specification Language for Mode-Based Systems

Joshua Bachmeier[1], Alexander Weigl[2], and Bernhard Beckert[2]
[1]FZI Research Center for Information Technology, j.bachmeier@fzi.de
[2]Karlsruhe Institute of Technology, Institute of Information Security and Dependability, {weigl,beckert}@kit.edu

Abstract

The first step in developing safe and functioning systems is the specification of the intended behavior. The development, validation, and verification depend on clear and unambiguous specifications. Building understandable specification tools requires adequate formalisms and representation to express the expected functional behavior. We present contract machines: a graphical specification language based on the well-known modeling concept of state machines and the intuitive semantics of assume-guarantee contracts. Contract machines (CMs) build upon the logical foundation of contract automata (CA) which are non-deterministic finite automata over alphabets of contracts, and provide the formal semantics of CMs. CAs can be processed by (semi-)automated verification and validation tools, such as model checkers or test case generators. In contrast to contract automata, contract machines offer a more high-level view of the system under scrutiny by providing more features to ease usability. We present features for effective controlling of non-determinism, using recurring specification patterns, e.g. for fault modes and error recovery behavior, and handling different versions and variants of systems.

1 Introduction

The verification of systems and software has been generally accepted by standards to provide high safety and security levels, e.g., ISO 26262 recommends highly the use of semi-formal or formal methods to ensure the safety of the two highest integrity levels. Nonetheless, using computer-assisted specification tools and formal methods for the specification creation is possible, although uncommon. [1, translated by the authors]. Design-By-Contract [2] is a well-established paradigm that states the behavior of components is described in terms of assumptions, guarantees, invariants, and side effects, which form its contract. In practice, however, writing such specifications is often not feasible – especially, for complex systems with different operation modes – due to the lack of understandable and accessible specification languages for system engineers.

In previous work [3], we introduce contract automata (CA), a formal specification language that can serve as a foundation or user-oriented specification formalism. CA can be used to specify the behavior of mode-based reactive systems. They take the form of finite-state machines and employ assume-guarantee contracts, to guard the transitions between modes. CA can be used to specify complex systems that exhibit different behaviors based on the *mode* that they are currently in. For example, a system might have the modes *starting*, *operational*, and *stopping*, each of which might need a different set of contracts to describe its behavior. The contracts serve the double purpose of describing the system behavior within a mode as well as guarding the transitions between the modes.

Compliance of a system to a CA is defined via game-based semantics, which can be understood as a more powerful variant of testing the system with symbolical values. Because of this semantics, CA can be translated into executable code which makes them applicable for a variety of verification techniques such as model checking, deductive verification tools, and runtime verification.

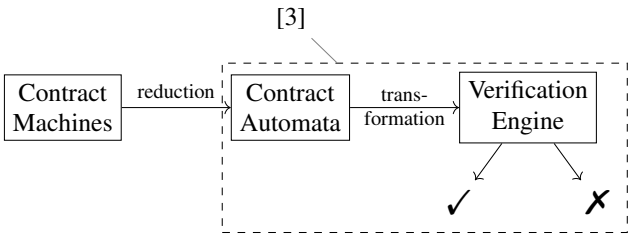

Figure 1 Verification Pipeline for CMs

Contribution CA were designed to have a simple theoretical foundation on the theory of formal languages and games. Therefore, they are also designed as a pure and mathematical construct to ease reasoning about them and simplify the implementation of verification systems. Nevertheless, they are very extensible and form the ideal foundation for building a more feature-rich and intuitive specification system.

In this work, we propose *contract machines (CMs)*: A formal and rigorously defined, yet usable, specification representation for reactive systems that maps to CA. CMs follow the graphical notation of state machines as introduced by UML and SysML but uses the the semantical foundation of CA. CMs are not simply a one-to-one graphical representation of CA. In addition to being a more engineer-oriented version of CA, they bring a set of comfort and convenience features to ease the system specification. This way CMs have the following advantages over plain CA:

1. They are more *intuitive* through their SysML-inspired notation and explicit handling of nondeterminism using ordering.

2. They increase *compartmentalization* and reduce *redundancy* using regions, a mechanism to group modes with common parts in their specification.

3. They make it easier to handle systems with *many modes* and *long contracts* utilizing their notation and the aforementioned regions.

4. They achieve better *integration* into real-world system life cycles and evolution, by providing explicit management of versions and variants.

The features, which are formulated as extensions, are handled by reducing a CM to its corresponding CA before verification or other subsequent tasks. This is possible because the extensions described in this work do not formally increase the expressive power of CA, they are only a matter of representation. Figure 1 illustrates the relation of CMs to CA and how they are placed in a verification pipeline. The transformation of a CA to a format suitable for a verification engine and the verification process (marked by the dashed box ⬚ in the figure) is covered in our preliminary work on CA [3].

Overview In Section 2, we give a summary of contract automata (CA), especially their syntax and semantics. Based on this, contract machines (CMs) are introduced in Section 4 including their extensions over CA. That section also presents the meta-model of CMs and how the extensions are reduced to a plain CA. Larger specification examples of CMs are given in Section 6 for a smart contract. We discuss extensions that we considered but ultimately did not include in CMs in Section 7. Finally, we conclude in Section 8.

2 Preliminary: Contract Automata

First, we give a note on terminology: We distinguish between *mode* and *state*. The state is a point in the potentially infinite space of all possible system configurations. For a software component, for example, this might be the product space of all possible variable values. A mode is an abstraction representing a set of states which are relevant in the system's specification. There are only finitely many modes to describe those distinctions in the state space which are relevant to the current analysis. What we call mode is the similar to what is called the enumerable *state variable* in typical UML state machines, while what we call state corresponds to UML's *extended state*. In this paper, we are mainly concerned with modes.

Example Let us consider a simple counter which periodically emits increasing or decreasing values between -128 and 128. The contract automata is given in Figure 2. The system has an input variable *enabled* of type Boolean and an output variable *val* of type integer. The CA expresses that, after the start of the system, the output value is first increased until reaching the maximum value is reached, and then the system decreases the output until it reaches the minimum value if the *enabled* is true. The contract automata has two modes, 'up' representing the mode of increasing values and 'down' for decreasing. The assumption in the contracts on the edges ensures that *val* is only changed when *enabled* is true. Whenever reaching ± 128, the specification changes the mode, and so does the applicable contract (counting direction). Note that the contract does not specify the exact amount which the value should be increment or decrement. A compliant implementation to the CA may add or subtract 1 or any other positive number in each cycle as long as the limits are adhered to. Also, an implementation that alters between the outputs $val = 128$ and $val = -128$ is also compliant.

Syntax We start with a set of *input* and a set of *output variables*. These variables can be used in *formulas*, which serve to express properties and relations of the input and output variables. A pair of formulas, labeled *assumption* and *guarantee* form a *contract*. In the previous example, *enabled* is the only input and *cnt* the only output variable. A contract is for example $\neg enabled / cnt = \text{old}(cnt, -1)$, where *negenabled* is the assumption and $cnt = \text{old}(cnt, -1)$ the guarantee. The term $\text{old}(cnt, -1)$ refers to the output value *cnt* which was present in the last operation.
In addition to the sets of input and output variables, a CA consists of a set of *modes*, an *initial mode* and the possible *transitions* between the modes. A transition connects to modes and is annotated with a contract. In the counter automaton, the modes are 'up' and 'down' and the transitions with the annotated contracts are visualized as edges in the graph.

Reactive System Analogously to a CA, a reactive system has input and output variables. For our verification pipeline (Figure 1), we assume it is deterministic. A system can be stateful meaning that it can store information and use that later in the computation of further output values. Apart from that, we make no further restrictions on its behavior and implementation. For the purpose of this paper, it can be regarded as a black box that iteratively receives an assignment of inputs and produces an assignment of outputs[1].

Semantics The semantics of a CA define the compliance of a reactive system against the CA. For this purpose, we consider the concept of *contract words*. A traditional finite automaton accepts a language of words made up of characters. In contrast, the CA accepts a language of contract words, which are sequences of contracts. A reactive system's behavior can be defined as a set of infinite words made up of input/output pairs, describing every possible run of the system.

[1] For static verification purposes, e.g. model checking, the actual implementation becomes relevant. However, this does not impair our ability to specify the behavior of black-box systems. Indeed, other techniques such as runtime verification or test case generation do not require knowledge of the inner workings of the system.

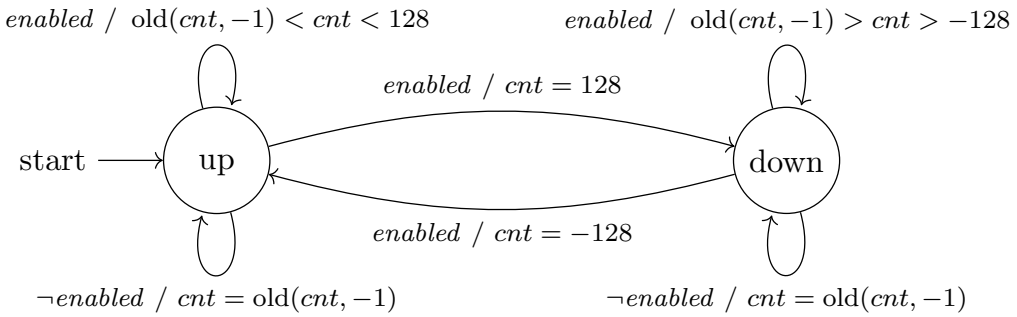

Figure 2 Contract automata describing systems counting up and down in $[-128, 128]$. Assumption and guarantees are separated with "/" on the transitions. old$(cnt, -1)$ refers to the output value *cnt* of the last operation.

We connect these concepts by matching the language contract word with a run of a system step by step, checking the current input/output pair against the possible contracts using the operational semantics of the automaton. Given a run of the system (sequence of input/output pairs), there are three possible outcomes on the CA. If there always exists an outgoing transition of an active mode, where the associated assumption and guarantee are fulfilled, then the run is *compliant*. For incompliant runs, we can distinguish two cases: (a) the run is *uncovered*, meaning the violation of a contract is due to the violation of assumptions, e.g. because the selected input values that are not valid in the current state; or (b) the run is *flawed*, meaning that a guarantee was violated because of a bad choice of output values by the system even though least one assumption is fulfilled.
With this high-level introduction of the central concepts of CA, we can define our user-friendly extensions and understand their semantics and how they map to the basic form of CA. For additional details and more precise and mathematically rigid definitions, we refer the inclined reader to our technical report [3].

3 Related Work

The use of state machines for system modeling and formal specification and verification has a long history, but is missing the use of contracts. In this section, we give a brief overview of selected state machine-based approaches.
Paltor and Lilius [4] present a formalization of the operational semantics of UML state diagrams to enable model checking. In contrast to our work, the state diagram is not used for behavioral specification and the UML semantics are covered completely. CMs are just using a subset of UML state diagram elements.
RoboChart [5], a "timed state-machine based formal notation for robotics", is closer to the original state machine notation by using entry and exit actions, and also using Boolean guards and state updates on transitions. Their semantics are based on process algebra.
Darvas et al. [6] presents the specification approach *PLC-specif* targeted at programmable logic controllers (PLCs). They build upon on a signature (similar to input and output variables and definitions), a set of defined events (similar transition guards), a state machine, an output functions from the current state to values and invariants over the internal state and the signature. Note that the specification is operational and deterministic by defining an executable Mealy automaton. CMs on the other hand have no operational semantics and may be nondeterministic.
Herrmannsdörfer et al. [7] provides a survey on modeling and specification approaches using tabular notation for state machines covering Parnas Tables and SCR. The latter got a revival as NuSCR [8]. Herrmannsdörfer et al. argues that tabular specification is advantageous over graphical methods, since they are easier to validate and more compact. CMs bring together the best of both worlds, by enhancing classical graphical automata with several usability features, making them more compact, while having a rigorous formal model in the background to allow validation and verification. We have previously explored table-based specification as Generalised Test Tables [9], which can be seen as a precursor to CA.
Itsykson [10] introduces a formalism to describe the behavior of software libraries using state machines. The behavior of the whole library, as well as of each created object, is specified by a separate state machine. Transitions between states are triggered by the library functions, which in turn can have pre- and postconditions. Transitions in CMs directly have assumptions and guarantees, and the fulfillment of those makes a transition possible.

4 The Graphical Representation of Contract Machines

Contract automata (CA) are very useful formalism for verification, but to make them usable by system engineers, further enhancement are required. While a natural visualization for a CA is obvious (transition diagrams) we take a more systematic approach to defining a graphical representation based on SysML and UML state charts. Additionally, CA as described so far are very bare-bones, and writing them for moderately large systems is already cumbersome. In this section, we describe contract machines (CMs): a user-friendly, versatile, and extensible graphical representation for CA. We present several extensions to the representation that can be naturally reduced to "plain" CA.

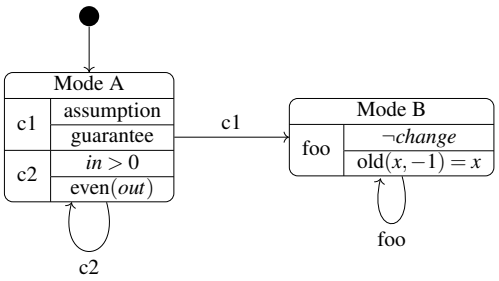

Figure 3 A basic CM using no extensions

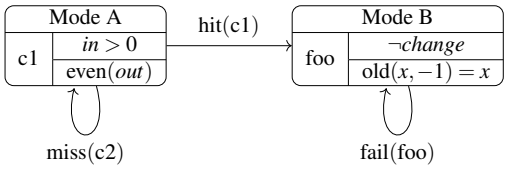

Figure 4 A CM using transition guards to refer to the contracts

4.1 Foundation: SysML State Machine

To lower the gap between the specification language and the perspective of the engineer, we borrow the notations from the state machines as defined by SysML and UML. In particular, we reuse the notation for states, transition, and sub-states. In contrast to UML, we have a well-defined meaning of a contract state-machine, by defining a reduction to a CA.

Figure 3 shows a simple example of a state machine representing a CA. This is the first step towards CMs. Note that this example is purely illustrative and does not specify and useful system. Firstly, we use the states and transition of a state machine with the same meaning as modes and transition in a CA. To increase readability, we write contracts inside the modes, assigning each one a label. On the transition, we refer to the contracts by their label (the contracts of the source mode are referable on each transition). To mark the start mode, we borrow the initial state notation from SysML.

4.2 Extension to the representation

With the state machine notation so far, we are already able to fully express CA. Besides the position of the contracts, it is a one-to-one mapping to a CA. To bring further convenience to a user of our specification methodology, we define further extensions to the visual representation.

4.2.1 Transition Guards

So far, the annotations on the transitions can only refer to contracts directly. This means that the transition is taken if and only if the current input/output pair adheres to the referenced contract, in other words, if the contract is *hit*. There exist two other meaningful cases: Either the contract is *missed* (meaning its assumption is violated) or that contract *fails* (meaning that the assumption holds, but the guarantee is violated).

	Contract	
Guard	Assumption	Guarantee
c	a	g
$\text{hit}(c)$	a	g
$\text{miss}(c)$	$\neg a$	true
$\text{fail}(c)$	a	$\neg g$

Table 1 Mapping rules of transition guards to plain contracts for a contract $c = (a, g)$.

Instead of guarding a transition by using a contract verbatim, which corresponds to a hit of the contract, we explicitly distinguish between the above cases, by annotating the transitions with $\text{hit}(c)$, $\text{miss}(c)$ or $\text{fail}(c)$, where c is a contract. The transition is taken, when the current input/output pair hits, misses, or fails the contract, respectively. If a contract c is used as a guard as-is, then $\text{hit}(c)$ is implied. Figure 4 shows how these *transition guards* can be used and their semantics. Note that the given examples shown here are syntactically valid but semantically meaningless. Table 1 gives an overview of how the guards are mapped to contracts. With this definition, reducing a CM with transition guards to a CA is trivial.

These operators allow to specify the behavior in unexpected or faulty situations. For example, the user can specify how a system should behave after an unspecified situation occurs.

4.2.2 Regions

When specifying systems using CA, certain patterns emerge, for example, some conditions are present in assumptions and guarantees and sometimes complete transitions are repeated, throughout a set of modes.

To accommodate this, we define *regions* as a mechanism to group modes together and attach a common set of behaviors.

Visually, a region is represented by a rectangle surrounding a group of modes. It might also surround other regions, which in turn contain modes. So formally, each mode and each region is associated with a parent region, except the top-level region. The structure thus forms a tree with the regions as inner nodes and modes as leaves.

These regions can carry a set of global contracts, which are implicitly added as available contracts to each mode in the region, as well as a global assumption and guarantee, which are implicitly added conjunctively to the assumptions and guarantees of all contracts in the region, regardless of whether they are defined in a region or directly in a mode.

Additionally to outsourcing common contracts or parts of contracts to the region, the user can also outsource common outgoing transitions: It is possible to define a transition from a region to a mode. The transition is guarded by a contract defined in this region, or any parent region. Semantically, this transition is then present in each mode contained in the region.

Figure 5 shows an example of a CM with a region. The region encompasses the modes *Mode A* and *Mode C*. The

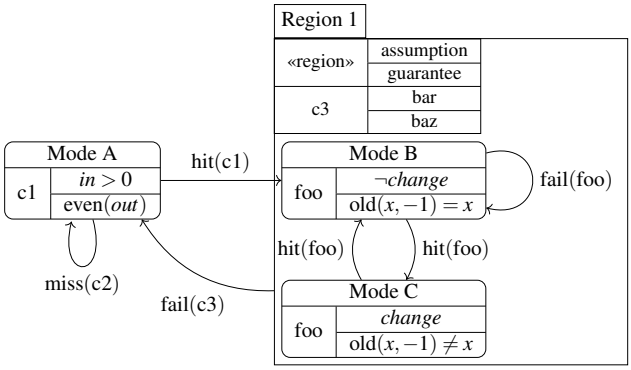

Figure 5 A CM using a region

global contracts of the region *Region 1* are written in its upper left corner. The mark «region» indicates the global assumption and guarantee of the region. We can see one region-wide transition from *Region 1* to *Mode A*.

4.2.3 Transition Precedence for Determinism

CA as a mathematical model are nondeterministic: It is trivially possible that a given input/output valuation satisfies more than one contract. If the guards of multiple outgoing transitions of the same mode are satisfied, all of these transitions may be taken and there are multiple successor modes. In this case, we call the guards (and contracts) *overlapping*, since the space of input/output valuations are not disjoint. In the specification of systems, this nondeterminism is often unintended and hence a source of specification flaws. If nondeterminism is intended, you should make it explicit. The manual way to disable this nondeterminism is by explicitly including the negation of the overlapping contracts in the contract itself. If you need to mutually exclude more than two overlapping contracts, the contracts get convoluted and hard to understand, even more, it is hard to figure out the precedence of transitions.

In CMs, we provide a way to control nondeterminism by ordering the outgoing transitions of each mode. The user assigns to each transition a numerical priority. The value of the assigned numbers determines a total order: lower numbers indicate a higher priority. If the guards of multiple transitions are satisfied, the one with the highest priority is taken. Only if two overlapping transitions have equal priority, there is true nondeterminism. Figure 6 shows how this is represented visually. In this example, the guards fail$(c1)$ and hit$(c2)$ overlap for example for $in = 7$ and $out = 5$. By assigning priority 1 to the upper, and 2 to the lower transition, we ensure determinism: In this case, only the upper transition miss$(c1)$ is taken. Under the hood, this is implemented by constructing new contracts that are mutually exclusive to all outgoing transitions (of the same mode) with a higher priority. Of course, the construction must respect the semantics of the different guards.

4.2.4 Conditional Transitions

One of the major challenges in the industry is the management of product families over time. We address this by supporting variants and versions using *conditionals* on

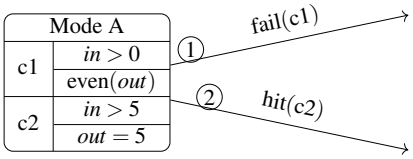

Figure 6 A partial CM using transition precedence

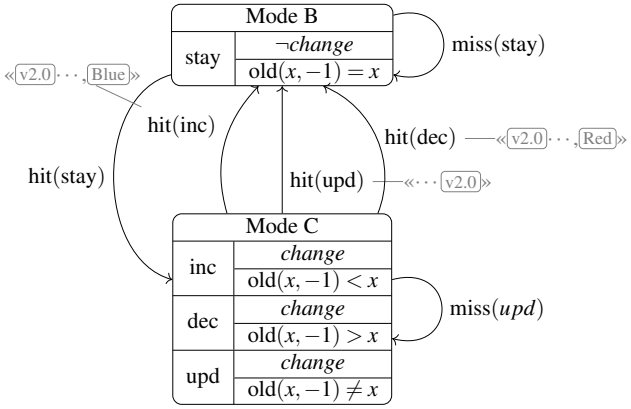

Figure 7 A CM using conditionals

transitions. These conditions allow the user to include or exclude a transition for a specific set of versions or variants. We unify the handling of versions and variants: We specify both symbolically, with a natural order to handle versions. For variants, we use symbol like foo or bar. For versions, we use a semantic versioning scheme, e.g., v3.0 or v0.2.3 with v3.0 > v0.2.3.

Each transition can optionally be annotated with a list of conditions. A condition can either be a plain symbol or a range of symbols. A range is written as v2.3 ··· v4.0, which means all versions from and including v2.3 upwards to and excluding v4.0. The range can be open-ended on either side as ··· v4.0 and v2.3 ··· to include *all* version up to or from that point onwards, respectively. If a mode or contract becomes unreachable, because all incoming transitions are excluded, it can be removed.

When reducing a CM to a CA, a set of current version and variant symbols must be provided. A transition is only included in the resulting CA when every condition in the conditional is satisfied by the current symbols. For plain symbol conditions, this trivially means that the symbol must be included in the set. For range conditions, this means that a symbol must be included in the set that is within the range, according to the natural order. If a transition has no or an empty conditional, the transition is always included into the effective CA.

Consider the example in Fig. 7. Here, the specified behavior was refined with the release of v2.0. Previously, it was merely required that activating the input *change* while in Mode C results in a change of the output variable x while transitioning to Mode B. Starting with v2.0, there are two variants: Red and Blue. In the red variant, the same transition is required to increase x, while in the blue variant, it must decrease.

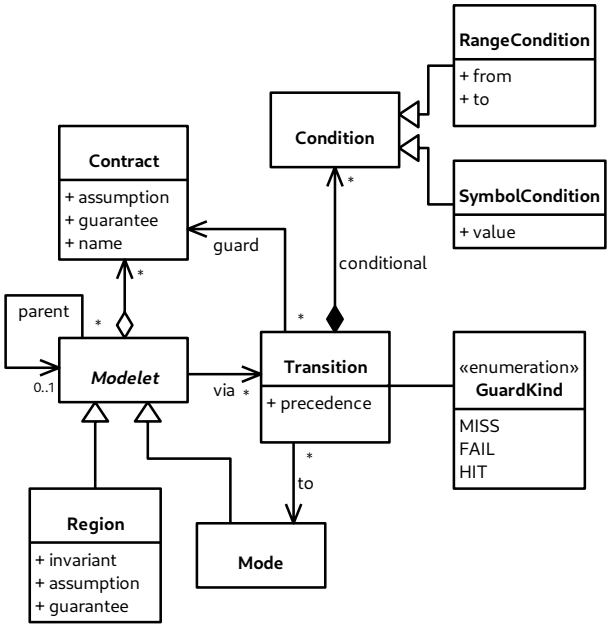

Figure 8 Metamodel of CMs

4.3 Metamodel

To give a comprehensive understanding of CMs, Fig. 8 shows the complete metamodel as a UML class diagram, including all the concepts discussed in this section. Firstly, a CM consists of Regions and Modes. They have some aspects, such as outgoing transitions and the parent relationship, in common. Therefore, they are derived from the abstract class Modelet: A Modelet can have a parent (modes can be contained in regions and regions can in turn be contained in regions). A Modelet might have no parent, in which case it is the top-level region. The Modelet carries the Contracts and we can go *via* an outgoing Transition *to* a concrete Mode. A Transition refers to a Contract as its *guard*, in combination with the GuardKind, which models the guard operators as an enumeration. In addition to the precedence, a transition has multiple Conditions as its *conditional*, which can be RangeConditions (for version ranges) or plain SymbolConditions.

4.4 Comparison with UML state diagram concepts

Both notation and meaning of CMs is borrowed heavily from UML state diagrams. Especially semantically, however, there are quite a few important conceptual distinctions. In this subsection, we highlight and discuss the most important of these differences.

Recall from Section 2 that the modes of CMs correspond to the state variable in UML. The UML extended state, the full internal state of the system, is not present in CM, since we regard systems as black boxes.

Fundamentally, UML state diagrams have an *execution semantics*, while CMs are about checking the compliance of systems. In other words, CMs can't be executed, since there are no actions specified, only contracts. In UML state diagrams on the other hand, the user specifies actions on the transitions, which determines the output. Transitions in CMs have guards, essentially made up of an assumption and a guarantee, which are formulas about the input and output variables. In UML, Transitions have triggers, and can have guards (which are different from CM guards) and actions. These two concepts can not be compared one by one. In UML, the trigger specifies the event that triggers the transition, while in CMs, the transaction is available, when the input/output pair satisfies both the assumption and the guarantee. The UML guards can be used to specify a formula that needs to hold for a transition to be taken, which makes them similar to CM assumptions. UML guards can talk about event parameters, which can be compared to the systems input variables used in CMs, and extended state variables, which have no correspondence in CMs.

Apart from these conceptual differences, there are some features of UML state machines which have no equivalent in CMs. For instance, we have no hierarchical nested states. In UML, this is a sub-machine contained within a state of the super-machine. Upon entering the state, the system also transitions into the starting state of the sub-machine and responds to events accordingly. This concept can be arbitrarily nested. The whole system's state variable is made up of the current states in all the sub-machines from the current point in the hierarchy up to the root. In CMs, the structural hierarchy is formed by the regions. Conceptually, they are different from hierarchical states, being used to hold contracts, transitions and formulas common to multiple modes. A similarity is that both regions and hierarchical states allow outgoing transitions that apply in all contained states/modes.

The term "region" also appears in UML state diagrams as part of orthogonal regions, which is a mechanism to represent product automata. While CMs, or rather CA, are closed under taking of the product, no mechanism to express this is included in the current version.

UML state machines have the notion of *history states*, which allows reentering a hierarchical sub-machine in the state from which it was left. In CMs, we do not maintain a history of modes and since we have no real sub-machine, the concept is not applicable. It is however possible to refer to previous input and output variables using the $\mathrm{old}(\dot{})$ operator.

5 Case Study: Automatic Emergency Brake System

For our first case study (Figure 9), we take the automatic emergency brake system of a car. The system is based on an older version of Mathworks' Simulink example "Autonomous Emergency Braking with sensor fusion"[2]. In our theoretical report [3], we specify the brake system using plain CA and are able to statically verify a C implementation using model checking.

[2] URL: `https://www.mathworks.com/help/driving/ug/autonomous-emergency-braking-with-sensor-fusion.html`, accessed September 2023

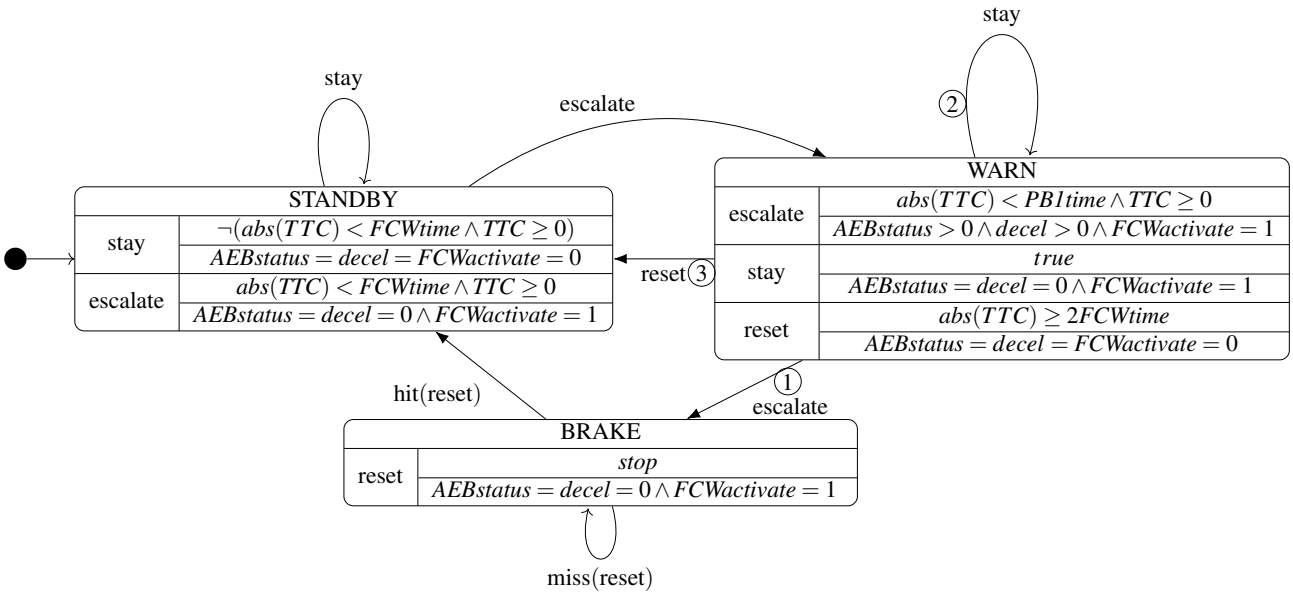

Figure 9 CM of an Assistant Emergency Brake System

The purpose of this system is to monitor the estimated time to collision (*TTC*, provided as input, can be calculated from the change in distance to the text car) and engage the brakes. The input variable *stop* indicates whether the car is in a standstill. The remaining input variables *FCWtime* and *PB1time* are constants setting the *TTC* threshold values to activate the collision warning and to engage the brakes. The outputs of the system are the deceleration *decel*, the status of the system *AEBStatus* and whether the collision warning should be active (*FCwactivate*).

The system starts in STANDBY mode, and must stay there, as long as the *TTC* is lower than the warning threshold *FCWtime*. The warning must of course be inactive and the deceleration zero. Only if the time to collision goes above this threshold, the warning indicator is activated ($FCWactivate = 1$) and the system escalates to mode WARN. In this mode, the default action is to maintain the status quo, realized by the contract *stay* with the trivial assumption true having the lowest priority. Otherwise, two options are possible: Either the *TTC* increases sufficiently which resets the system to STANDBY mode, or *TTC* decreases further, going below the next threshold *PB1time*, leading to an escalation to BRAKE mode and to a deceleration, i.e. $decel > 0$. In the BRAKE mode, the *stay* contract once again handles the default case while the *reset* contract hits only once the vehicle comes to a full standstill. It then transitions back to STANDBY mode.

This case study illustrates the usefulness of CMs for a realistic system component. We use guards and transition precedence to simplify the specification.

6 Case Study: Casino Smart Contract

In this case study, we specify a smart contract that implements a simple Casino [11], which implements a guessing game for head or tail of a coin toss. Figure 10 shows the CM. In contrast to the previous case study, a smart contract implements an interactive system where a caller invokes the action (incl. specific parameters) that the system should perform. The contract names correspond to these action names. This case study demonstrates how the CM concept can be applied to a very different use case. The contracts in this case are not simple assume-guarantee-pairs, but follow the definition of solc-verify [12] contracts, which have not only pre- and post-conditions but also *modifies* clauses (abbreviated as **mod** in the figure). The modifies clause allows the user to specify which variables are allowed to change their value. Any variable v that is not part of the modifies clause, must keep its value, i.e., $v = \text{old}(v)$. The `msg.sender` and `msg.balance` are also implicit constants representing the identification of the caller and its balance.

We use a region to define common contracts for the actions `addToPot`, which moves money from the casino operator's wallet to the internal pot and `remFromPot`, which does the opposite. The Casino starts in mode *idle*, allowing `addToPot` and `remFromPot`, and the creation of a play.

`createPlay` is called by the operator with their choice of HEAD or TAIL, secured by a cryptographical bit commitment scheme, such that the choice can not be deduced from the public state. The `createPlay` contract takes a parameter n. In the CM this is just a normal input variable that is unused in all other cases, the "(n)" can be seen as part of the contract name. If a play is available, anyone can challenge the operator by guessing its choice and placing a bet. The action `decideBet` transfers the double amount of the bet to the challenger if the guess was correct (contract `decideBetWin`). Otherwise, the money is retained (`decideBetLoose`) and can be withdrawn by the operator (contract `removeFromPot`). Either way, the operator can start another play afterward.

A peculiarity of smart contracts is their public nature: They run on a ledger or blockchain architecture, such that their

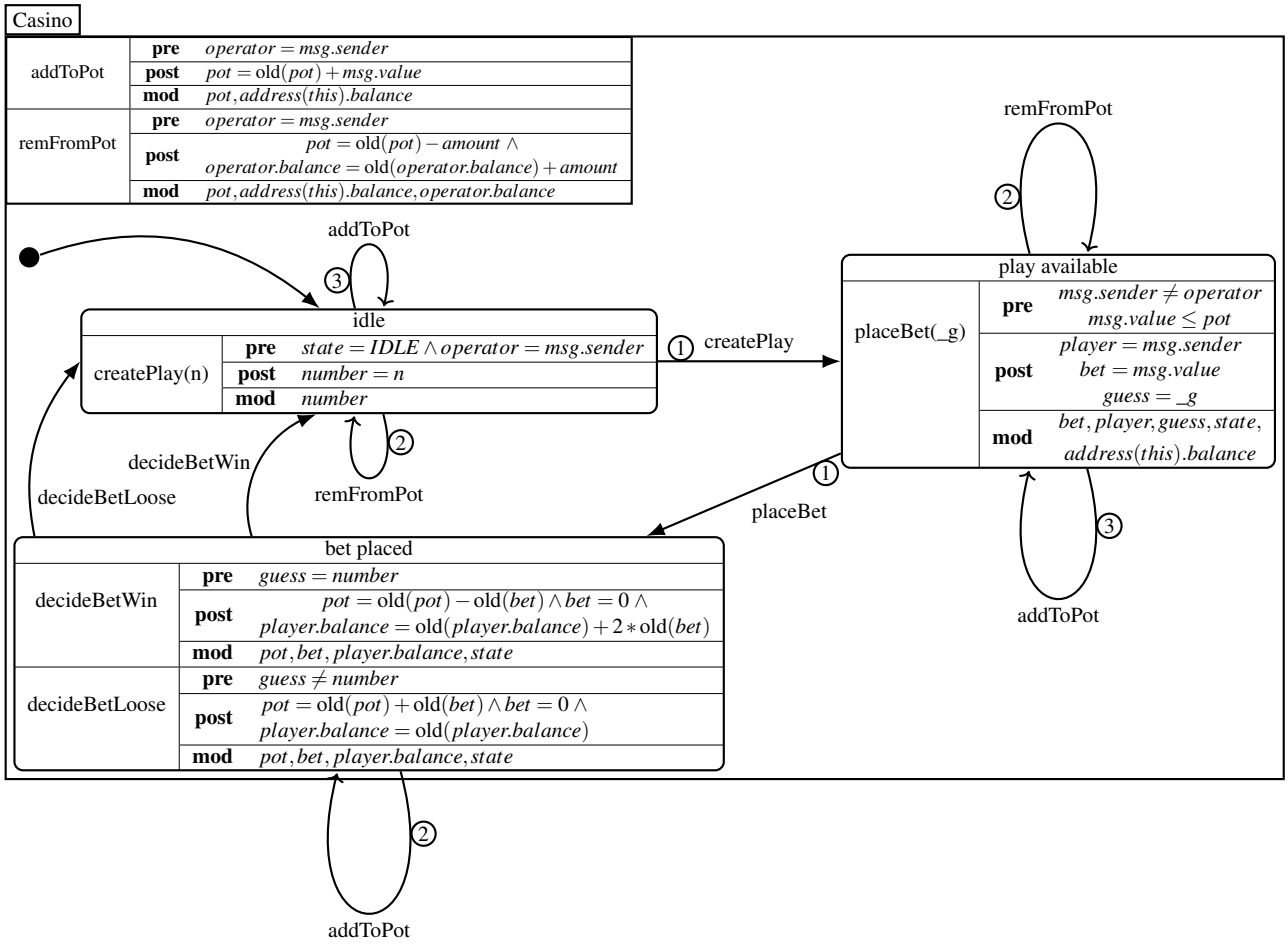

Figure 10 CM of a smart contract realizing a guessing game on a blockchain. Contracts to correspond to possible interactions with the smart contract.

code and data are disclosed. A consequence of this is the special need for robustness even with malicious users. A CM only describes the behavior provided that the assumptions of the contracts hold. The casino CM for example only describes the behavior positively, the so-called "happy path". To handle malicious environments, we may want to close the set of possible actions, i.e. by adding an implicit loop transition on each mode with the contract "n : true/true/$\emptyset$" with the lowest precedence n, a pre- and post-condition true and no modification of the state variables (empty modifies-clause). This would ensure that the system is not affected by unspecified events.

7 Omitted Extensions and Future Work

CMs as presented here are already versatile and feature-rich specification mechanisms. Nevertheless, they can be extended even further. In this section, we briefly discuss some possible extensions that we considered, but finally rejected. Some of these might be realized in the future, when it can be shown that their usefulness justifies the increased complexity. Furthermore, we want to continue to explore the usefulness of CMs before adding more features. To this end, an implementation of a graphical modeling tool for CMs is currently being worked on. We already discussed some omitted features in Section 4.4, we do not repeat those here.

Guard expressions In the current version of CM, transition guards can refer to only a single contract, wrapped in one of the guard operators hit, miss or fail. A more general approach would be to allow arbitrary expressions over contracts including the operators and a set of Boolean connectives, such as the logical operators and, or, not etc. Guards might then look like this: $\text{miss}(c_1) \wedge (\text{hit}(c_2) \vee \text{hit}(c_3))$. Note e.g. that $\neg\,\text{hit}(c)$ would not be the same as $\text{fail}(c)$, but rather identical to $\text{fail}(c) \vee \text{miss}(c)$. This would allow for more flexibility and expressiveness when specifying transitions, at the cost of increased complexity of the metamodel and the reduction to CA.

Mode Inheritance We considered mode inheritance or template modes as a generalization of regions. This would allow specifying modes as a derivation of other modes inheriting their contracts and transitions. This concept serves the purpose of grouping common properties of multiple modes and making it reusable across the diagram. We ultimately settled on regions as the most visual approach, since

it makes the grouping of the related modes clear in the graphical representation. This is especially true for outgoing transitions, which would be less obviously associated with the concrete modes were they attached e.g. to template modes.

Conditionals for Regions We introduced conditionals as a way to include transitions only in certain variants or versions of the CM. This approach can naturally be extended to modes and regions as well. An open question is how the exclusion of a mode affects incoming transitions, which should be included according to their conditional. In the current version, to exclude modes the user has to explicitly set the conditionals to exclude all incoming transitions.

Region transition loops It is already possible to attach outgoing transitions to regions, which are applied to all contained modes. However, the target of a transition must always be a concrete mode. For the general case, it is unclear what the semantics of a transition going to a region should be, since there is no starting mode in a region and even more so, the modes contained in a region do not necessarily form a sub-machine. It is a common pattern in CMs to have identical or similar "loop transitions" on many modes (e.g. *stay* in Section 5). For this case – a loop transition on a region – the semantics seem much clearer: Apply the loop transition to all modes in the region.

8 Conclusion

In this paper, we present contract machines (CMs), a graphical mechanism to specify the behavior of complex model-based systems. CMs are a user-friendly extension of contract automata (CA), which form the theoretical foundation of our work. They are based on the established specification utility of state machines, as they can be found e.g. in UML or SysML, paired with contract-based verification. We introduce and discuss extensions to the graphical representation: *Transition guard* to flexibly reference contracts, *regions* to reduce redundancy and group common aspects of parts of the specification, *transition precedence* to explicitly handle nondeterminism and *conditionals* to manage different variants and versions of the specification. With these extensions, CMs become a intuitive and usable specification tool with a solid formal foundation that can serve as a catalyst for a variety of validation and verification use cases. We further explore this by presenting the CM specification of two small case studies, an automated emergency brake system and a smart contract casino scenario. We plan to continue to evaluate and expand our methods in the future, by implementing tools and using these to specify larger and more complex systems.

References

[1] M. Hauke, M. Schaefer, R. Apfeld, T. Bömer, M. Huelke, T. Borowski, K.-H. Büllesbach, M. Dorra, H.-G. Foermer-Schaefer, J. Uppenkamp, O. Lohmaier, K.-D. Heimann, B. Köhler, H. Zilligen, S. Otto, P. Rempel, and G. ReuSS, "IFA Report 2/2017, Funktionale Sicherheit von Maschinensteuerungen –Anwendung der DIN EN ISO 13849–," Institut für Arbeitsschutz der Deutschen Gesetzlichen Unfallversicherung (IFA), Sankt Augustin, Tech. Rep., Apr. 2017.

[2] B. Meyer, "Applying "design by contract"," *IEEE Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[3] A. Weigl, J. Bachmeier, B. Beckert, and M. Ulbrich, "Contract automata: A specification language for mode-based systems," Karlsruhe Institute of Technology, Tech. Rep., 2023, released on 16.12.2023. [Online]. Available: https://doi.org/10.5445/IR/1000164563

[4] I. Paltor and J. Lilius, "Formalising UML state machines for model checking," in *«UML»'99: The Unified Modeling Language - Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 28-30, 1999, Proceedings*, ser. Lecture Notes in Computer Science, R. B. France and B. Rumpe, Eds., vol. 1723. Springer, 1999, pp. 430–445. [Online]. Available: https://doi.org/10.1007/3-540-46852-8_31

[5] P. Ribeiro, A. Miyazawa, W. Li, A. Cavalcanti, and J. Timmis, "Modelling and verification of timed robotic controllers," in *Integrated Formal Methods - 13th International Conference, IFM 2017, Turin, Italy, September 20-22, 2017, Proceedings*, ser. Lecture Notes in Computer Science, N. Polikarpova and S. A. Schneider, Eds., vol. 10510. Springer, 2017, pp. 18–33. [Online]. Available: https://doi.org/10.1007/978-3-319-66845-1_2

[6] D. Darvas, E. Blanco Viñuela, and I. Majzik, "A formal specification method for PLC-based applications," in *Proceedings of the 15th International Conference on Accelerator and Large Experimental Physics Control Systems*, 2015, pp. 907–910.

[7] M. Herrmannsdörfer, S. Konrad, and B. Berenbach, "Tabular notations for state machine-based specifications," *Crosstalk*, vol. 21, no. 3, pp. 18–23, 2008.

[8] J. Cho, J. Yoo, and S. D. Cha, "Nueditor - A tool suite for specification and verification of nuscr," in *Software Engineering Research, Management and Applications, Second International Conference, SERA 2004, Los Angeles, CA, USA, May 5-7, 2004, Selected Revised Papers*, ser. Lecture Notes in Computer Science, W. Dosch, R. Y. Lee, and C. Wu, Eds., vol. 3647. Springer, 2004, pp. 19–28.

[9] B. Beckert, M. Ulbrich, B. Vogel-Heuser, and A. Weigl, "Generalized test tables: A domain-specific specification language for automated production systems," in *Theoretical Aspects of Computing - ICTAC 2022 - 19th International Colloquium, Tbilisi,*

Georgia, September 27-29, 2022, Proceedings, ser. Lecture Notes in Computer Science, H. Seidl, Z. Liu, and C. S. Pasareanu, Eds., vol. 13572. Springer, 2022, pp. 7–13. [Online]. Available: https://doi.org/10.1007/978-3-031-17715-6_2

[10] V. M. Itsykson, "Formalism and language tools for specification of the semantics of software libraries," *Autom. Control. Comput. Sci.*, vol. 51, no. 7, pp. 531–538, 2017.

[11] "Casino smart contract challenge." [Online]. Available: https://verifythis.github.io/02casino/

[12] Á. Hajdu and D. Jovanovic, "solc-verify: A modular verifier for solidity smart contracts," in *Verified Software. Theories, Tools, and Experiments - 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13-14, 2019, Revised Selected Papers*, ser. Lecture Notes in Computer Science, S. Chakraborty and J. A. Navas, Eds., vol. 12031. Springer, 2019, pp. 161–179. [Online]. Available: https://doi.org/10.1007/978-3-030-41600-3_11