Contract Automata: A Specification Language for Mode-Based Systems

Alexander Weigl
weigl@kit.edu
Karlsruhe Institute for Technology
Germany, Karlsruhe

Joshua Bachmeier
j.bachmeier@fzi.de
FZI Research Center for Information Technology
Karlsruhe, Germany

Mattias Ulbrich
ulbrich@kit.edu
Karlsruhe Institute for Technology
Karlsruhe, Germany

Bernhard Beckert
beckert@kit.edu
Karlsruhe Institute for Technology
Karlsruhe, Germany

ABSTRACT

The comprehensive, understandable and effective formal specification of complex systems is often difficult, especially for reactive and interactive systems like web services or embedded system components. In this paper, we propose *contract automata*, a new specification formalism for describing the expected behaviour of stateful systems. Contract automata combine two established concepts for formal system specification: contract-based specification and nondeterministic finite state automata. Contract automata restrict the effects that the operations of the specified system may have using input-output-contracts. The automaton structure of a contract automaton describes when contracts are applicable. Contract automata support the refinement and composition of reactive systems, enabling modular verification of systems assembled of multiple subsystems. In this paper, we formally define the semantics of contract automata based on a two-party game between the system under test and its environment. We define the proof obligations and present techniques to prove a refinement relationship between contract automata, the validity of system compositions, and the compliance of source code against a contract automaton. We provide a tool for the generation of the proof obligation that can be discharged with model-checkers or static program analyses. We exemplify the use of contract automata by presenting the specification and verification of an emergency brake assistant.

ACM Reference Format:
Alexander Weigl, Joshua Bachmeier, Mattias Ulbrich, and Bernhard Beckert. 2024. Contract Automata: A Specification Language for Mode-Based Systems. In *Formal Methods in Software Engineering (FormaliSE) (FormaliSE '24), April 14–15, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3644033.3644381

FormaliSE '24, April 14–15, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0589-2/24/04
https://doi.org/10.1145/3644033.3644381

1 INTRODUCTION

Motivation. Design-By-Contract [18] is an established software engineering paradigm in which the admissible behaviour of software modules is described by formal contracts. A contract for an operation specifies the required properties of the environment and input values as a precondition, and the guarantees on the produced output values as a postcondition. The presence of formal contract specifications for the operations of a system enable static and modular formal verification. The necessity of specifying contracts for all operations is the main disadvantage of the Design-by-Contract paradigm. Specifications need to be abstract enough to cover multiple implementations, but strong enough to show the relevant properties of the subsystems and ultimately the complete system.

While logical formulas are mostly sufficient to state assumptions and guarantees for individual operations, they lack expressiveness when it comes to specifying *reactive systems* that interact with their environment. In a reactive system, assumptions and guarantees for the same operation may vary from call to call: Reactive systems are stateful, i.e., their output in response to an input may depend on the internal system state and thereby on previous inputs. Hence, a specification language suitable for reactive systems should also support stateful specifications. For example, Linear Temporal Logic allows stateful specifications implicitly through its temporal operators. In many cases, the succession of admissible operations follows the defined protocol of the system. The admissible input-output behaviour – i.e., the applicable contract – is the same for many states within the same step of the protocol. Which assumptions and guarantees hold in a state, hence, often depends only on a small aspect of the state that captures the current *mode* of the system (within the protocol). Tracking such protocol steps or modes can be done naturally by applying the concept of automata.

Contribution. We propose *contract automata* – a specification language for mode-based interactive and reactive systems. Contract automata are nondeterministic finite automata over assume-guarantee contracts. Both formal concepts, automata and assume-guarantee-contracts, have been widely and successfully used for the specification of system behaviour. Previously, the two approaches have not yet been conceptually integrated. The automata aspect of contract automata is used to describe the possible modes of the system and the possible transitions between them. Assume-guarantee-contracts are used to constrain the transitions, i.e., the

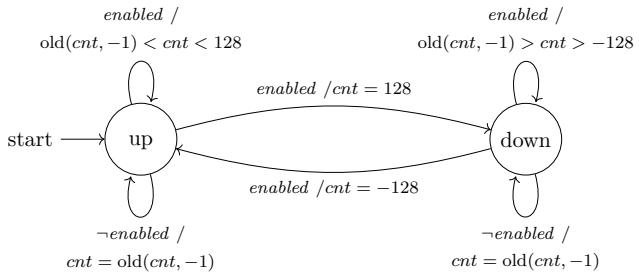

Figure 1: Contract automaton describing systems counting up and down in [-128,128]. Assumption and guarantees are separated with "/" on the edges. $\text{old}(cnt, -1)$ **refers to the output value** *cnt* **of the last operation.**

possible operations in different modes. These modes may reflect the internal state of the system, but they can also reflect the state of the environment. The idea behind contract automata is to track the current situation in the environment and in the system as the modes. This allows the specification of the expected inputs of the environment and expected responses of the system in each of these particular situations. The relationships between the input and output are defined symbolically by formulas.

Figure 1 shows the contract automaton for a system with the input variable *enabled* (Boolean) and the output variable *val* (integer). The specification expresses that the value may first only be increased until reaching the maximum value 128, and then only decreased until reaching the minimum value −128. The contract automaton has two modes, one for increasing (up) and one for decreasing (down) *cnt*. The contracts on the edges make sure that *cnt* is only changed when *enabled* is true. Whenever reaching ±128, the specification changes its mode and therefore the applicable contract (i.e. the counting direction). Note that the automaton does not specify the amount to increment or decrement. A compliant implementation may add or subtract 1 or any other positive number in each cycle as long as the limits are adhered to.

The main application focus of contract automata lies on reactive and interactive systems. These have in common that they maintain an internal state, evolve over time, and are meant to run for an indefinitely long period.

Advantages. Contract automata are ideal for modular and refinement-based formal analyses of systems. In this scenario, a system module is initially specified using a contract automaton on abstract properties of the system. This abstract contract automaton is then used together with contract automata of other modules to modularly verify that the interaction between the subsystems is correct on this abstraction level (e.g., by conducting a protocol verification using model checking). This verification abstracts from details about the implementation, but these can be added later using refinements. Later, when implementing the system module, the contract automaton is refined to a more specific contract automaton, a composition of subsystem contract automata, and eventually to an implementation. As long as any refinement follows the Liskov substitution principle [16], the implementation maintains the safety guarantees established on the abstract contract automaton.

Another advantage of contract automata is that they form an ideal basis for building an Engineer-friendly, feature-rich and expressive graphical specification language. In [2], we reuse this foundational framework to build Contract Machines – a formal graphical specification language building upon the established notion of state-machines with more high-level features., e.g., support for version and variants to cover multiple product lines.

The concept of contract automata is an adaptation of Generalised Test Tables (GTTs) [4] for a broader application. GTTs are a formal specification based on *test tables*, a concept widely used in industry to describe a time series of test stimuli. GTTs aim to be comprehensible and usable by industry practitioners. A drawback of GTTs is their limited specification coverage as the behaviour for only a single family of similar test cases is described. We lift this restriction and proposes contract automata as a foundation for stateful specification that cover the complete system behaviour. It adds formal foundations for refinement and modular reasoning.

Overview. In this paper, we define the syntax and semantics of contract automata (Section 2), and explain their encoding into a transition system for static verification with model checking and program verification tools (Section 3).

We explain the necessary proof obligations required for the modular verification in Section 4, following the principles established by Cimatti et al. [10]. We provide support for the verification of C programs against contract automata, for refinement and the validity of the system composition of contract automata. We demonstrate the usage of contract automata on a component of an emergency braking system for cars (Section 5). The tool and the evaluation are publicly archived [19].

2 FORMAL FOUNDATIONS OF CONTRACT AUTOMATA

2.1 Syntax of Contract Automata

Contract automata are used to describe the input-output behaviour of reactive systems. For the specification, the two sets I and O contain the input and output variables of the specified system. In this paper, we assume that constraints over the input and output values are formulas in $Fml_{I,O}$, the quantifier-free predicate logic over the theories of integers and reals. Additionally, the term $\text{old}(x, -n)$ can be used to refer to the previous value that the variable $x \in I \cup O$ held $n \in \mathbb{N}$ system iterations ago. In particular, $\text{old}(x, 0)$ refers to the current value of x and $\text{old}(x, -1)$ to the previous value. Both the assumption and the guarantee are evaluated after the system produced the output. Typically, the assumption talks about the input values whereas the one can contain both input and output variables.

The alphabet used to label transitions in contract automata are formal contracts which are pairs $c = a_c/g_c \in Fml^2_{I,O}$ consisting of an assumption (also called precondition) a_c and a guarantee (also called postcondition) g_c. We use the functions $\text{assume}(c) = a_c$ and $\text{guarantee}(c) = g_c$ to access the assumption and guarantee components of the contract c. Let $C_{I,O} = Fml^2_{I,O}$ denote the set of contracts over I and O. The indices are skipped when clear from context. With the formal concepts of contracts introduced, contract

automata can now be formally defined as nondeterministic finite automata (NFA) over contracts:

Definition 2.1 (Contract Automata). A contract automaton $A = (I, O, M, M_0, \delta)$ for input variables I and output variables O consists of a non-empty finite set M of modes, a non-empty set of starting modes $M_0 \subseteq M$ and a transition relation $\delta \subseteq M \times C_{I,O} \times M$.

In this paper, the states of a contract automaton are called *modes* to avoid confusion with the notion of states in the transition system and the reactive system implementation itself. Modes are specification entities which allow one to abstract from the full state space of the specified systems. Note that the definition does not comprise accepting modes since there is no need for the distinction between accepting and non-accepting modes. However, for the upcoming definition of the language of a contract automaton, every mode in M can be considered accepting. In the following, the sequence of contracts along finite walks in the automaton is relevant:

Definition 2.2 (Paths of a Contract Automaton). For a contract automaton $A = (I, O, M, M_0, \delta)$, the set $\text{paths}_C(A) \subseteq C^*$ of finite paths in A is the regular language $L(A) \subseteq C^*$ that A accepts when interpreted as a NFA over the alphabet C in which every mode $m \in M$ is considered an accepting state.

Hence, any path is obtained by beginning in a start mode $m \in M_0$ and following an arbitrary number of transitions in δ while collecting the traversed contracts in C from the transition annotations. A path may terminate in any mode (every mode is accepting), hence, it is obvious that the set $\text{paths}_C(A)$ is closed under prefix, i.e., if $p \in \text{paths}_C(A)$, then every prefix of p is also in $\text{paths}_C(A)$.

2.2 Semantics of Contract Automata

Contract automata are used to specify reactive systems. To define the semantics of contract automata, we first must give a formal definition of reactive systems. A reactive system responds to an input (assignment of variables in I) with an output (variables in O) respecting and maintaining its current internal state. The sets I and O contain the syntactical *variables* for in- and output. Let $\mathcal{I}$ and $\mathcal{O}$ denote the possible input and output *values*. For paths in contract automata, we considered *finite* walks in the automaton, but since a reactive system may run forever we must define its semantics as a set of *infinite* input-output traces. The upcoming definition of reactive systems is restricted to deterministic systems, since this aligns with the targeted application scenario. A non-deterministic system can still be modelled in this setup by adding extra input variables that encode the nondeterministic choices as external choices. Due to the universal nature of the notion of compliance (Definition 2.5), the specification must hold for all possible values of these extra variables and thus for all possible nondeterministic choices.

Definition 2.3 (Reactive System). A response function $R : \mathcal{I}^* \to O$ of a reactive system assigns to every input history the deterministic output response. The behaviour $\mathcal{B}(R) \in (\mathcal{I} \times \mathcal{O})^\omega$ of R is the set of *infinite input-output traces* with

$$\mathcal{B}(R) = \left\{ \left((i_0, o_0), (i_1, o_1), \dots \right) \mid R(i_0, \dots, i_n) = o_n \text{ for } n \in \mathbb{N} \right\} .$$

When evaluating a formula $f \in \mathit{Fml}$ in a finite sequence $\bar{\sigma} = (\sigma_1, \dots, \sigma_n) \in (\mathcal{I} \times \mathcal{O})^n$ of input/output value pairs, the value of a (input or output) variable is taken from the most recent tuple σ_n. In Section 2.1, the operator $\text{old}(\cdot, \cdot)$ was introduced as a construct in *Fml* to reference values from the observed history. In the above evaluation context, the expression $\text{old}(x, -k)$ is the evaluation of variable x in the tuple σ_{n-k}. The remainder of the formula evaluation is standard predicate logic (with fixed interpretations according to the supported theories). We write $\bar{\sigma} \models f$ to denote that a formula is satisfied by the finite sequence $\bar{\sigma}$ of input/output values.

For an infinite trace $\tau \in \mathcal{B}(R)$ and a path $\bar{c} \in \text{paths}_C(A)$ (i.e., a finite sequence of contracts) with $|\bar{c}| = n$, we write $\tau \models \bar{c}$ to denote $\bigwedge_{i=1}^{n}(\tau_1, ..., \tau_i) \models \text{assume}(c_i) \wedge \text{guarantee}(c_i)$, i.e. that every contract in $\bar{c}$ is satisfied by the respective partial trace of τ. We write $\tau \models \text{assume}(\bar{c})$ to say that all assumptions of $\bar{c}$ are satisfied on the trace.

After defining the semantic notion of reactive systems, we define when a reactive system is *compliant* to a contract automaton. Intuitively, we want to capture the fact that whenever a system is presented with a sequence of inputs that adhere to the assumptions of the then applicable contracts, the system response must also adhere to the then applicable contract guarantees. It is the automaton structure which decides which contracts are applicable in which historical context. This structure makes this a non-trivial definition.

Moreover, the definition must cover a gap between the notions defined for contract automata and for reactive systems: For a contract automaton A, the set of contract words $\text{paths}_C(A)$ is a sequence of contracts – which are purely syntactical entities. In contrast, the behaviour of a system $\mathcal{B}(R)$ is a set of infinite input-output traces over the value spaces $\mathcal{I}$ and $\mathcal{O}$ – and thus, a semantic concept. To bridge this gap, we define when a single input-output trace $\bar{\sigma}$ is compliant with a path of contract automaton A: If a contract word $\bar{c} \in \text{paths}_C(A)$ that is satisfied by the input-output trace can be extended by a single contract c' such that the assumptions $\text{assume}(\bar{c} \cdot c')$ are satisfied, where $\cdot$ is concatenation of contract (traces), then there must also be a continuation of the trace c'' (not necessarily the same as c') that satisfies all the assumptions *and* the guarantees.

Definition 2.4 (Trace Compliance). Let R be a reactive system and A be a contract automaton with the same input and output variables. An input-output trace $\tau \in \mathcal{B}(R)$ is *compliant* to A if

$$\begin{aligned} &\forall \bar{c} \in \text{paths}_C(A).\ \tau \models \bar{c} \\ &\quad \implies \left(\exists c' \in C.\ \bar{c} \cdot c' \in \text{paths}_C(A) \wedge \tau \models \text{assume}(\bar{c} \cdot c')\right) \\ &\quad \implies \left(\exists c'' \in C.\ \bar{c} \cdot c'' \in \text{paths}_C(A) \wedge \tau \models \bar{c} \cdot c''\right) \end{aligned}$$

R is called compliant with A if every $\tau \in \mathcal{B}(R)$ is compliant with A.

The compliance of a reactive system w.r.t. a contract automaton can equivalently be described by the two-party game outlined in Figure 2 as a pseudo code algorithm. The game is played between the *environment* (as the challenger) generating the input values and the *system* responding (by executing the underlying program code). The possible moves each of the two "players" can make are determined by the contract automaton: The environment must adhere to the assumptions and the system must be compliant to the guarantees. The first player to not leave any possible moves to continue the game loses the play. Therefore, the optimal strategy for the environment is to select input values $\bar{i} \in \mathcal{I}^*$ which adhere to

Input: Reactive System $R : \mathcal{I}^* \to \mathcal{O}$ and
contract automaton $A = (I, O, M, M_0, \delta)$
Output: Returning the winner (system or environment)
$\hat{M} \leftarrow M_0$; $\bar{\sigma} \leftarrow \varepsilon$; $\bar{\imath} \leftarrow \varepsilon$; $k \leftarrow 0$;
while *true* **do**
// Invariant: $\hat{M} = \{m \mid m_0 \xrightarrow{\bar{c}} m, m_0 \in M_0, \bar{c} \in \text{paths}_C(A), |\bar{c}| = k, \bar{\sigma} \models \bar{c}\}$
Environment chooses input $i \in \mathcal{I}$;
$\bar{\imath} \leftarrow \bar{\imath} \cdot i$;
System computes response $o = R(\bar{\imath}) \in \mathcal{O}$;
$\bar{\sigma} \leftarrow \bar{\sigma} \cdot (i, o)$;
$\hat{E} \leftarrow \{(m, c, m') \in \delta \mid \bar{\sigma} \models \text{assume}(c) \wedge m \in \hat{M}\}$;
if $\hat{E} = \emptyset$ **then**
return *System wins*; // Chosen input not covered by active contracts
end
$\hat{M} \leftarrow \{m' \mid (m, c, m') \in \hat{E} \wedge \bar{\sigma} \models \text{guarantee}(c)\}$;
if $\hat{M} = \emptyset$ **then**
return *Environment wins*; // Output violates all possible contracts
end
$k \leftarrow k + 1$;
end

Figure 2: Game between environment and system R w.r.t. the contract automaton A.

contract assumptions and trigger the system to produce an output that violates the guarantees. The set $\hat{M} \subseteq M$ represents the current configuration (i.e. the modes that are reachable given the history observed so far) of the contract automaton. Initially, $\hat{M}$ contains the starting modes. In each iteration of the play, the environment is first asked for an input value and then the system for its response.

The algorithm in Figure 2 chooses the input values from the environment nondeterministically. This means that, given infinite plays of the game, *every* possible choice of values from the environment will be considered, and all of this must be correctly handled by the system for it to be compliant. The system wins a play when the environment violates the assumptions. This captures the fact that such a play is considered to be out of the scope of the specification. Since the variable $\hat{M}$ holds the set of possible modes, we simulate the power automaton of the given contract automaton and thereby resolve the nondeterminism in the contract automaton. Unlike the environment, the system does not have any degree of freedom, its responses are determined entirely by the underlying program. In each round (iteration of the body of the while loop), three outcomes are possible:

(1) There are no outgoing transitions from any mode in $\hat{M}$ such that the finite trace $\bar{\sigma}$ satisfies the assumption ($\hat{E} = \emptyset$). In this case, the system wins since the environment chose an input not covered by any specification.
(2) Amongst the outgoing transitions $\hat{E}$ that satisfy the assumption, no mode also satisfies the corresponding guarantee. In this case, the environment wins since the system response violates the guarantees of all remaining possible transitions.
(3) After the update, $\hat{M}$ is not empty, i.e., there is at least one mode reachable (under $\bar{v}$) that satisfies both the assumption and the guarantee. In this case, the play continues for another round with the updated history.

Definition 2.5 (Game compliance). A reactive system R is called *game-compliant* to a contract automaton A if there is *no* play of the game in Figure 2 that ends with the result "Environment wins".

This definition of game compliance considers all possible choices of the environment, and therefore all possible input-output traces in $\mathcal{B}(R)$ are covered. The game (Definition 2.4) and the trace compliance (Definition 2.5) are equivalent.

Theorem 2.6. *A reactive system R is game-compliant with a contract automaton A if and only if R is trace-compliant with A.*

The claim can be be proven by contraposition: R is not trace-compliant with A if and only if there exists a play won by the environment. The structure of the game in Figure 2 operationalises the condition described by the formula in Definition 2.4. This allows us to establish a direct correspondence between a witness for the quantifiers and a state of a play of the game. To this end, first look at the loop invariant of the algorithm as annotated in the pseudocode: $\hat{M}$ contains precisely those modes that are terminal modes in the language construction of Definition 2.2 for a contract path $\bar{c} \in \mathit{paths}(A)$ with $\sigma \models \bar{c}$ of length $k = |\bar{c}|$. This entails that $\tau \models \bar{c}$ ensures that the play with the inputs provided by τ spans at least k rounds. This is because the prefixes of $\bar{c}$ are all witnesses of the fact that in each round the set $\hat{M}$ was not empty and, hence, the conditional guards $E = \emptyset$ and $M = \emptyset$ could never have been satisfied for these inputs.

Assume that R is not trace-compliant. Then, from the negation of the condition in the definition, there are $\tau, \bar{c}, c'$ with (1) $\tau \models \bar{c}$, (2) $\tau \models \text{assume}(\bar{c} \cdot c')$ and (3) $\tau \not\models \bar{c} \cdot c''$ for any continuation $\bar{c} \cdot c''$ of c. Thanks to (1) the play for the inputs from τ has at least $|\bar{c}|$ rounds (as elaborated above). In the $(k+1)$-th round the first conditional is not triggered; E is not empty because of (2). However, (3) makes sure that the second condition is indeed triggered as there is no possible continuation of $\bar{c}$ that satisfies both assumption and guarantee. The converse argument is quite analogous, and from a play terminating in "Environment wins", one can construct witnesses that violate the condition in Definition 2.4.

3 CONTRACT AUTOMATA AS TRANSITION SYSTEMS FOR VERIFICATION

To enable formal verification of contract automata, we encode the game semantics of a contract automaton into a symbolic transition system. We define the encoding in a simple form. Backreferences, i.e. the old-operator are added later in Section 3.2. When bundled with an encoding of the system software to be verified, this allows us to discharge compliance proof obligations using model checkers and C-program verification tools. For this, the transition system for a contract automaton models the progress of the game and decides the winner of a play. The modes of the automaton are represented by Boolean variables. A dedicated variable encodes that the system loses the game. An invariant is used to specify that this mode must

not be reachable. Since the game internally stores all reachable modes (and not a single reached mode), the transition system for a contract automaton A essentially encodes the power automaton. The transition system's state contains Boolean mode variables to encode the currently active modes of A in the game. Additional mode variables are added to indicate the violation of assumptions and guarantees. The implemented approach produces specifications in the SMV format, which can be understood by many state-of-the-art model-checking tools; as well as the C programming language to allow verification with program verification tools.

3.1 Encoding of the Symbolic Transition System

The symbolic transition system is a logical encoding of the game in Figure 2. A symbolic transition system $TS = (\Delta, Init, Trans)$ is a tuple, consisting of a signature Δ (a set of variables), a formula $Init \in Fml_\Delta$ over the symbols in Δ to describe the initial state, and a formula $Trans \in Fml_{\Delta \cup \Delta'}$ describing the transition relation between states. Since the transition connects two states, the formula refers to variables in the current state (Δ) and in the next state (Δ' where all symbols are primed).

In the following, we construct a transition system $TS_A = (\Delta_A, Init_A, Trans_A)$ for a given contract automaton $A = (I, O, M, M_0, \delta)$ to determine whether a system is game-compliant with A. For each automaton mode $m \in M$ there is a Boolean variable $\mathfrak{s}_m \in \Delta_A$. There are two additional Boolean variables $\mathfrak{s}_\bot, \mathfrak{s}_\top \in \Delta_A$ for assumption and guarantee violation. Δ_A also contains the input and output variables I and O of the system under specification:

$$\Delta_A = I \cup O \cup \{\mathfrak{s}_m \mid m \in M\} \cup \{\mathfrak{s}_\bot, \mathfrak{s}_\top\}$$

The variable $\mathfrak{s}_m$ is the symbolic representation of the automaton mode m. $\mathfrak{s}_m$ is true if and only if the automaton mode m is active in the play, i.e., if $m \in \hat{M}$. The variables $\mathfrak{s}_\top$ and $\mathfrak{s}_\bot$ are true if and only if all possible assumptions or a guarantee were violated in the most recent step, respectively. There exists only one initial state in the transition system in which the original starting modes M_0 of the contract automaton are active. The variables $\mathfrak{s}_\top$ and $\mathfrak{s}_\bot$ are initially false.

$$Init_A = \left(\bigwedge_{m \in M_0} \mathfrak{s}_m \right) \wedge \left(\bigwedge_{m \notin M_0} \neg \mathfrak{s}_m \right) \wedge \neg \mathfrak{s}_\top \wedge \neg \mathfrak{s}_\bot$$

$Trans_A$ encodes the mode transitions of the contract automaton A and also the cases in which all applicable assumptions or guarantees are violated by the environment or system. The cases in which the assumptions are violated are relevant for refinement (Section 4).

$$\begin{aligned} Trans_A = \bigwedge_{j \in M} \Bigg(\mathfrak{s}'_j \leftrightarrow \bigvee_{(i,c,j) \in \delta} \mathfrak{s}_i \wedge \text{assume}(c) \wedge \text{guarantee}(c) \Bigg) \\ \wedge \mathfrak{s}'_\top \leftrightarrow \mathfrak{s}_\top \vee \bigvee_{(i,c,j) \in \delta} \mathfrak{s}_i \wedge \neg\, \text{assume}(c) \\ \wedge \mathfrak{s}'_\bot \leftrightarrow \mathfrak{s}_\bot \vee \Bigg(\bigvee_{(i,c,j) \in \delta} \mathfrak{s}_i \wedge \text{assume}(c) \wedge \neg\, \text{guarantee}(c) \\ \wedge \bigwedge_{o \in M} \neg \mathfrak{s}'_o \Bigg) \end{aligned}$$

Since failing assumptions or guarantees can be flagged using the variables $\mathfrak{s}_\bot$ and $\mathfrak{s}_\top$, the relation $Trans_A$ is a total relation in the inputs and outputs in the sense that for every state and every input-output value pair, there is a successor state.

The invariant $Inv = \neg \mathfrak{s}_\bot$ states that the system has always adhered to at least one of the available guarantees. This situation corresponds to the winning condition of the system (in Definition 2.5).

To verify the compliance of a system R with the CA A, we combine the transition system TS_R, which represents the behaviour of the system R faithfully, with the TS_A of A using the product system $TS_R \parallel TS_A$. The product system is itself a symbolic transition system defined as $TS_A \parallel TS_B = (\Delta_A \cup \Delta_B, Init_A \wedge Init_B, Trans_A \wedge Trans_B)$. Note that the product system combines both transition systems into a single transition system that simulates a synchronous execution of both systems under the condition that the predicates $Init_A$ and $Init_B$ do not restrict the domains of the same variables. Formally, the formula $Init_A \wedge Init_B$ must have the same models in the signature Δ_A as $Init_A$. The analogue holds for $Trans$.

Theorem 3.1. *A system R is game-compliant with a contract automaton A if and only if $TS_A \parallel TS_R \models \square Inv$.*

The transition system TS_A is a model of the game, which is evaluated on the input of the environment and outputs of TS_R. Similarly, TS_R is a model of the system R. Theorem 3.1 states that the parallel composition of TS_A and TS_R should never violate the invariant, i.e. that $\mathfrak{s}_\bot$ never becomes true. The invariant Inv is violated if and only if the system R is not game-compliant, meaning that there is an execution which leads to a state where no automaton modes are active due to the violation of assertions. The transition system constructed for a contract automaton restricts only the values of the automaton mode variables, not on the input, output or other variables of the system TS_R. Therefore, $TS_A \parallel TS_R$ remains a faithful model of the system R

3.2 Extension for backreferences

So far, the transition system TS_A does not support the backreferences expressed by the old-operator. To support these, we enrich the transition system, especially $Trans$, to maintain the history of each variable $v \in I \cup O$. In practice, we limit the history only to those variables and indices that occur as arguments to the old-operator in the specific contract automaton A. Let v be an input or output variable of the contract automaton and k the highest back-reference index. We then add the state variables $v_1, \ldots, v_k$ to the state of the transition system, and strengthen $Trans$ with $v'_1 = v \ \wedge \ \bigwedge_{i=2}^{k} v'_i = v_{i-1}$. In the initial state, these history variables would be under-specified. For practical reasons, we initialize the history variables with a default value to make the behaviour deterministic. This makes counterexamples more understandable.

4 REFINEMENT AND SYSTEM COMPOSITION

Reactive systems are typically modularly constructed as a composition of subsystems. This compositional nature can be exploited for formal analysis. In program analysis, the design-by-contract paradigm is *the* enabler for modular verification techniques by abstracting from subsystem behaviour with contracts, thus decomposing a verification task into one task per subsystem. To enable

modular verification for reactive systems with contract automata, two formal notions are important and must be adapted to contract automata: (a) *Composition:* What is the (most precise) contract automaton of a composed system if (only) the contract automata of the subsystems are known, and (b) *Refinement:* when does a (compositional) contract automaton comply with the given contract automaton of the analysed system? Additionally, the "base case" – verification of a system implementation against an individual contract – must be possible.

For the composition of reactive systems, we employ the definition from the OCRA tool [5] in which a module's interface is given by a set of input and output ports. The input ports are controlled by the environment, and the output ports by the module itself. The internals are given in the form of a graph of subsystems with their interconnection of the input/output ports. In our approach, we extend the OCRA model such that one can either define a system by using system composition or implement the system with a program fragment written in a programming language. In the example in Section 5, the C programming language is used. Such composed system models can be found in more complex manifestations in the real world, e.g. composed using Function Block Diagrams in IEC 61131-3 or the Lingua Franca [17] model of reactive systems.

Definition 2.3 gives us the semantics of a reactive system. For the composition, we need to define reactive systems structurally. We need to distinguish between composited systems and "leaf systems" which are implemented in source code. The latter one cannot be structurally analysed.

Definition 4.1 (Reactive System Structure). The structure of reactive system R is given by (I, O, P), where I is a set of input and O a set of output variables (ports). For a composed system, $P = (V, E)$ is a direct acyclic graph, where the subsystems are the vertices V and the edges connect the system ports. For a leaf system, $P = (\Delta, \mathit{Init}, \mathit{Trans})$ is a transition system with $I, O \subset \Delta$.

The implementation $P = (V, E)$ for a composed system R forms a graph. The set V contains the subsystems of R and the edges E describe the connections between the input and output variables of the subsystems. For $(s, o, s', i) \in E$ where $s, s' \in V$ with $s = (I_s, O_s, P_s)$ and $s' = (I_{s'}, O_{s'}, P_{s'})$; $o \in O_s$ is an output variable of s, and $i \in I_s$ an input variable of s'. This can be read as: The output variable o of s is connected to the input variable i of s'. The connections in E define the information flow within the system. By that, they determine the order in which the subsystems are evaluated. In the following, we assume that the composition is non-circular, and hence an order of the execution of the subsystems exists (the topological order).

To sequentially compose two systems $A = (I_A, O_A, P_A)$ and $B = (I_B, O_B, P_B)$, we write $A \overset{oi}{\rightarrowtail} B$ to denote that the output of system A is forwarded to the system B, where $oi \subseteq O_A \times I_B$ defines how the output variables of A map to the input variables of B (cf. [14]). Note that $A \overset{oi}{\rightarrowtail} B$ is itself a system:

$$(A \overset{oi}{\rightarrowtail} B) = (I_A \cup (I_B \setminus \mathrm{rng}(oi)),\ O_A \cup O_B,\ P_B \circ oi \circ P_A)$$

where $\mathrm{rng}(oi)$ describes those inputs of I_B mapped to output from O_A (i.e., they are not free). $A \overset{oi}{\rightarrowtail} B$ offers the same inputs as A and the non-occupied inputs of B, provides outputs $O_A \cup O_B$ of both systems. Executing systems in parallel, denoted by $A \parallel B$, is a special case of sequential composition with $oi = \emptyset$. For example, consider Figure 3 which is the graphical representation for the composition

$$D = (A \parallel B) \overset{\{o_1/a, o_3/b\}}{\rightarrowtail} C\,,$$

where $A = (\{i_3\}, \{o_3\}, P_A)$, $B = (\{i_1\}, \{o_1\}, P_B)$ and $C = (\{a, i_2, b\}, \{o_2\}, P_C)$. The composited system D has three inputs $\{i_1, i_2, i_3\}$ which are not occupied internally, and all outputs of the subsystems $\{o_1, o_2, o_3\}$.

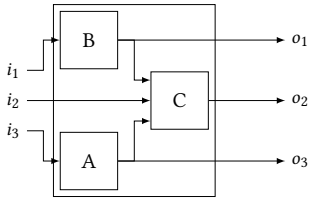

Figure 3: Example of the sequential composition.

For the verification, we express the composition $A \overset{oi}{\rightarrowtail} B$ in terms of transition systems. Let TS_A and TS_B be the corresponding transition systems to A and B, then $TS_{A \overset{oi}{\rightarrowtail} B} = (\mathit{Init}_A \wedge \mathit{Init}_B,\ \mathit{Trans}_A \wedge (\mathit{Trans}_B)\mu)$, where μ is a substitution of the input variables of B with the output variables of A according to oi. More precisely, $\mu\colon in \mapsto out'$ for all $(in, out) \in oi$. The output variables are primed to insert the output values out which are computed using the in.

4.1 Refinement

Refinement is a relationship between two systems. We say a system A is a refinement of system B if it is more concrete. From another perspective, refinement is also a process of clarification that tightens the possible behaviour. We define refinement in our context between contract automata, in which every specified behaviour of A must exists in specified behaviour of B. The refinement between contract automata corresponds to the Assume-Guarantee (AG) refinement in [9], which implies implementation containment. Intuitively, the subsystem A must at least accept the admissible input values of B, and must only produce output values that B can also produce. We allow A to extend the interface of B, i.e., it may use more variables. The refinement relationship then only covers the variables in both contract automata. New behaviour on newly introduced inputs or outputs is allowed and is not covered by the proof obligations.

Definition 4.2 (Contract Automata Refinement). Let $A = (I^A, O^A, M^A, M_0^A, \delta^A)$ and $B = (I^B, O^B, M^B, M_0^B, \delta^B)$ be contract automata where the signature of A extends the signature of B: $I^B \subseteq I^A$ and $O^B \subseteq O^A$.

(1) A contract c^A for A refines a contract c^B for B, written $c^A \preceq c^B$, if $\models \mathrm{assume}(c^B) \rightarrow \mathrm{assume}(c^A)$ and $\models \mathrm{guarantee}(c^A) \rightarrow \mathrm{guarantee}(c^B)$.
(2) A path $\bar{c}^A \in \mathrm{paths}_C(A)$ refines a path $\bar{c}^B \in \mathrm{paths}_C(B)$, written $\bar{c}^A \preceq \bar{c}^B$, if

$$|c_A| = |c_B| \quad \text{and} \quad c_i^A \preceq c_i^B \ \text{ for all } 1 \leq i \leq |c_B| \ .$$

(3) A contract automaton A refines a contract automaton B, written $A \preceq B$, if

$$\forall\, \bar{c}^A \in \text{paths}_C(A).\ \exists\, \bar{c}^B \in \text{paths}_C(B).\ \ \bar{c}^A \preceq \bar{c}^B \ .$$

The refinement of individual contracts Definition 4.2.1 is the Liskov substitution principle (LSP) [16], which is lifted to traces and then automata. According to the LSP, assumptions may be weakened and guarantees may be strengthened. This ensures that every individual operation which adheres to a contract in B must also adhere to the corresponding contract in A. For paths, the extension means that every position must adhere to the LSP. Hence a contract trace $\bar{c}^A$ is a valid substitute for $\bar{c}^B$, and we can formulate the lemma:

Lemma 4.3. *Let A and B be two contract automata with $A \preceq B$, then if a reactive system R is compliant with A, then R is also compliant with B.*

We can also reuse the previous conversion of contract automata into transition systems to verify the refinement relationship:

Theorem 4.4. *Given two contract automata A and B and their corresponding transition systems TS_A and TS_B, then:*

$$A \preceq B \iff$$

$$TS_A \parallel TS_B \models \square\left(\left(\bigwedge_{x \in V} x^A = x^B\right) \rightarrow (\neg\mathfrak{s}_\top^B \rightarrow \neg\mathfrak{s}_\top^A) \wedge (\mathfrak{s}_\bot^A \rightarrow \mathfrak{s}_\bot^B)\right)$$

where $V = (I^A \cap I^B) \cup (O^A \cap O^B)$ are the common variables.

The theorem follows from Definition 4.2 and the fact that that TS_A and TS_B model the games of the contract automata. The implication arises from Point 1 in Definition 4.2. For any identical inputs and outputs to transition systems TS_A and TS_B, if TS_A rejects the input, then TS_B also needs to reject the input ($\mathfrak{s}_\top$) vice versa for the output.

4.2 Composition

With Liskov's notion of refinement available, the verification conditions can now be formulated. There are multiple proof obligations: For each subsystem in the composition, we show that its assumptions are met by its environment. The relevant part of its environment is given by the guarantees of previously executed subsystems (defined by the connection between an output variable and the corresponding input variable), and the assumptions of the contract for the overall composition. Moreover, we need to show that the complete composition adheres to the guarantees of the composition contract. Note that these proof obligations imply a refinement relation (Definition 4.2) between the composition contract and the contract composition of all subsystems.

Definition 4.5 (Composition Validity). Let R_0 be a composed system, which is given as a composition $Sub = (R_1 \overset{oi_1}{\rightarrowtail} R_2 \overset{oi_2}{\rightarrowtail} \cdots \overset{oi_{n-1}}{\rightarrowtail} R_n)$ in topological order, and let TS_i be the transition system of the contract automaton for each R_i. The transition system TS_{Sub} represents the composition Sub: $TS_{Sub} = TS_1 \parallel \cdots \parallel (TS_n)\mu_{oi_{n-1}}$. The composition is valid if and only if for each R_i ($1 \leq i \leq n$):

$$TS_0 \parallel (TS_{sub})\mu \models \square\left(\left(\neg\mathfrak{s}_\top^0 \wedge \bigwedge_{j=1}^{i-1} \neg\mathfrak{s}_\bot^j \wedge \neg\mathfrak{s}_\top^j\right) \rightarrow \neg\mathfrak{s}_\bot^i \wedge \neg\mathfrak{s}_\top^i\right)$$

and lastly

$$TS_0 \parallel (TS_{Sub})\mu \models \square\left(\left(\neg\mathfrak{s}_\top^0 \wedge \bigwedge_{j=1}^{n} \neg\mathfrak{s}_\bot^j \wedge \neg\mathfrak{s}_\top^j\right) \rightarrow \neg\mathfrak{s}_\bot^0\right) \ .$$

to show the final adherence with the composition contract. The variables $\mathfrak{s}_\bot^i$ and $\mathfrak{s}_\top^i$ refer to the $\mathfrak{s}_\bot$ and $\mathfrak{s}_\top$ variables of the TS_i. The application of substitution μ is required to map the inputs and outputs of the composed system R_0 to the inputs and outputs of the subsystems Sub.

The connection between the subsystems $R_1, \ldots, R_n$ is handled by the given construction of TS_{sub}. The connectivity of the input and output variables of the composed system to the subsystems is then handled by the substitution μ. In general, the definition describes that proving the composition validity is reduced to showing that the contract automata of each subsystem behaves correctly ($\neg\mathfrak{s}_\bot$) and that the assumptions are met ($\neg\mathfrak{s}_\top$) under the assumption that the contract automata of all previously executed subsystems are adhered to. Special is that we can assume the assumptions of the overall system composition ($\neg\mathfrak{s}_\top^0$), but we also need to ensure that its guarantees ($\neg\mathfrak{s}_\bot^0$) are ensured by the guarantees of the subsystems.

5 CASE STUDY: EMERGENCY BRAKE ASSISTANT

System. Assistants for emergency braking (AEB) become mandatory in newly registered cars in 2024. The AEB decides whether the car should be slowed down or even stopped to prevent a collision with a vehicle (or object in general) in front of the car. For this, the AEB computes the time needed to bring the own car (called *ego*) to a full stop and checks this against the time to collision with the preceding vehicle (called *mio*). The computation of time-to-collision and stopping time is based on the velocities, the distance, and system parameters. This information is created by a fusion of radar, camera sensors and tracking systems. Both systems are outside of the investigated system boundaries. We concentrate on the escalation behaviour. The AEB system holds a state to realise a progressive escalation: When a collision becomes possible, the system warns. If a warning does not trigger a different driver behaviour and the collision becomes more likely, the system starts with partial braking until it further escalates and finally triggers full braking to reach a full stop.

Figure 4 shows the AEB system, which consists of four function blocks for the time-to-collision, stopping calculation, escalation logic and a comparison for recognising the stop of the car. The AEB is based on an older version of Mathworks' Simulink example "Autonomous Emergency Braking with sensor fusion"[1]. We re-implemented it in our system model using C code and the data types integer (32-bits) and Boolean.

The function block `TTCCalculation` receives the distance and velocity to the mio car. Note that the `mioVelocity` is relative to the `egoVelocity` and reflects the change of the distance to the mio car. Its main output is `TTC`, which is the time it would take for the ego car to hit the mio car at current speed. Independently, the stopping time is calculated by `StoppingTimeCalculation` which provides

[1] https://www.mathworks.com/help/driving/ug/autonomous-emergency-braking-with-sensor-fusion.html, accessed December, 2023

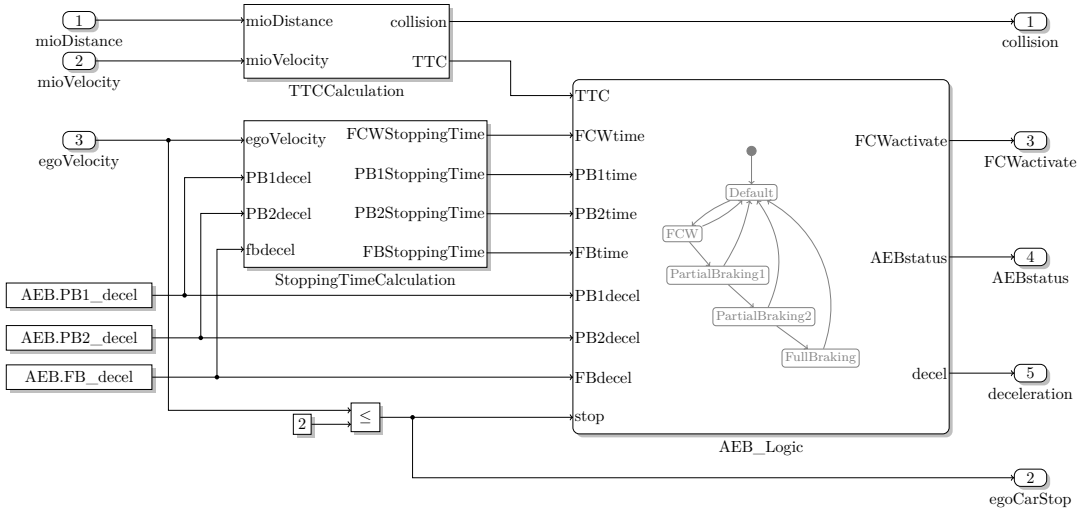

Figure 4: The graphical system description of the assistant for emergency brake. The variable prefixes PB and FB represent partial and full braking. This system is inspired by the *Autonomous Emergency Braking* example of Mathworks.

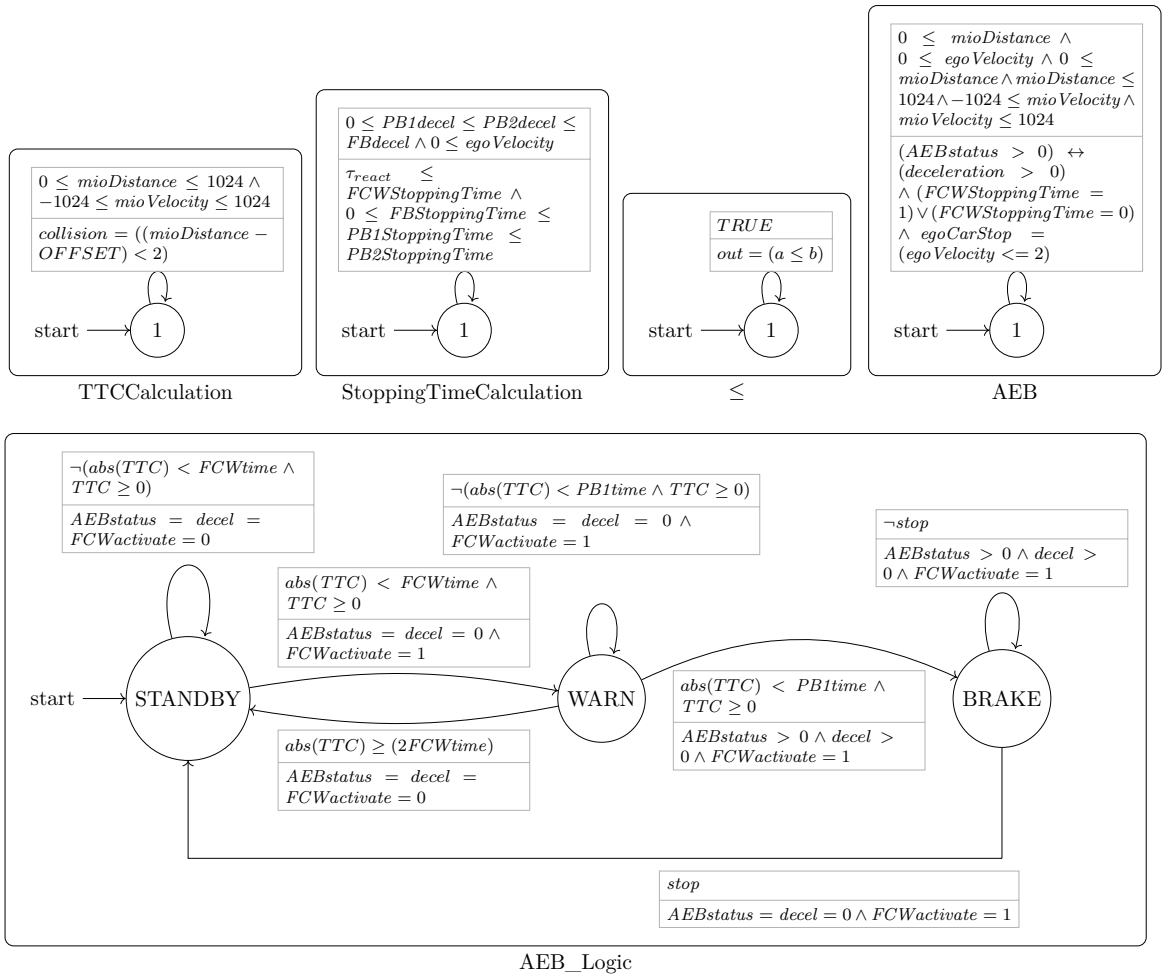

Figure 5: Contract automata for the emergency braking system

the time to stop under different deceleration strengths (forward collision warning, partial braking 1 and 2, and full braking). It also considers the latency of a human recognising the critical situation. Then the function block `AEB_Logic` combines all the calculated values and chooses one of the escalation levels. Escalation happens if the situation becomes worse, i.e., the difference between stopping time and time to collision decreases. De-escalation either requires a full stop of the car if the AEB decelerated the car, or if only a warning was given a decrease of the current ego velocity.

Contracts. The function blocks `TTCCalculation`, `StoppingTimeCalculation`, and less-than "≤" are stateless. Their behaviour can be described using a single assume-guarantee pair. For this reason, their automata have only a single mode in Figure 5. The contract automaton for `TTCCalculation` only specifies the input ranges of `mioDistance` and `mioVelocity` and specifies the computation of the `collision` output variable. The contract automaton for `StoppingTimeCalculation` is similar, it defines an order of the deceleration constants (partial deceleration is less than full deceleration), and guarantees the same order on the braking times. τ_{react} is the reaction time of the driver. Only `AEB_Logic` holds a state internally (cf. Figure 5). `AEB_Logic` handles the escalation of the system, the corresponding contract automaton only distinguishes between three modes (standby, warn, and brake) whereas the implementation has multiple modes for braking. The `AEB` contract automaton is the top-level CA of the complete system.

Tools. We implemented the prototype CAGEN[2] which generates the proof obligations required to fully prove a system definition consisting of the contract automata, the system and each subsystem, as well as the description of the composition structure. CAGEN generates proof obligations in SMV and C code. Proof obligations which are only based on contract automata, like refinement, and composition validity, are discharged by a model-checker using the SMV format. For this, the contract automata are translated into a transition system and the provided invariants are checked (Definition 4.5 and Theorem 4.4). The proof obligations that represent the compliance of an implementation against a contract automaton are written in C code and can be discharged using the C implementation of the subsystem. For this, we translate the transition systems of the contract automata into C code instead of SMV. This code would also be useable for runtime verification.

We use CBMC 5.50.0 [11] with KISSAT 3.0.0 for bounded verification (Bound $k = 256$), and for unbounded verification of C programs SeaHorn 10.0.0-rc0-3bf79a59 [12] with `-crab` (generation of invariants using abstract interpretation). Proof obligations in SMV are discharged with nuXmv 2.0.0 [7] and the IC3 [6] backend for invariants and LTL formulas. The companion artifact [19] provides the complete environment including the generated proof obligations.

Results. The results of the verification of the AEB are reported in Figure 6. Shown is CPU time in seconds as a median of three samples measured on an Intel Core i7-8565U with 16 GB memory. We need to discharge nine proof obligations: We verified each of the four implementations of `TTCCalculation`, `StoppingTimeCalculation`, ≤, AEB_Logic against their contract automata (Figure 5). We also need to prove the validity of the composition, including four checks of the adherence to the assumptions for each subsystem and one check of top level system against its contract. For the sake of completeness, the verification of our introductory example (Figure 1) takes 87 seconds with CBMC, and 2 seconds with SeaHorn.

[2]available under https://github.com/wadoon/cagen and in the artifact [19]

	nuXmv	CBMC	SeaHorn
≤	–	0.37	0.13
TimeToCollision	–	3.76	0.14
StoppingTimeC.	–	2.47	0.12
AEB_Logic	–	9.53	1.14
Comp. Validity	5.69	–	–

Figure 6: Performance (cpu time in seconds, median of three runs) of discharging the proof obligations for the AEB system, and its sub-components.

6 RELATED WORK

Automata and state machines are a common mechanism to describe the behaviour of systems. The combination of automatons with contracts does appear in the literature, albeit in a different fashion than contract automata.

In [3], a different concept also named *Contract Automata* is used for the behavioural specification of services. These Contract Automata have a focus on the orchestration of services. Therefore, the edge labels in the automata define requests, offers or idle actions of the services to be specified, which corresponds to the invocation of methods. Compatibility is defined as the actions playing well together and the end of behaviours being reached. This concept is very common, for example, interface input/output automata [15] also describe the emitted output and expected inputs as events. In contrast, our notion focuses more on the assumption and guarantees of a system and represents them as formulas describing the expected relation between input and output values. Moreover, Chilton et al. [9] present a framework for assume-guarantee and refinement that is well applicable to the refinement of contract automata given in Section 4. They focus on the required proof obligations in terms of the encoding as a transition system. A different contract automata approach is presented by Azzopardi et al. [1] which is similar to [3]. In their automata concept, each state is associated with a set of clauses. Each clause belongs to a party. On the edges, you write the events which lead from one state to the next. The clauses are contracts over obligations (a party must or must not do an action) and permissions (a party is allowed to do an actions). The authors also introduce *contract (automata) strictness* which resembles the refinement idea. In comparison, our approach is also applicable to more than two parties, as we are agnostic toward the internal structure and meaning of the contracts. For didactic reasons, we present the special case that a contract is pair of pre- and postcondition – the former belongs to the environment and the latter belongs to the system. The subjects of the contracts from Azzopardi et al. are single events, while our contracts evaluated about whole input-output traces. In terms of the analyses, the focus of Azzopardi et al. [1] lies on runtime verification and conflict detection.

CoCoSpec [8] is a specification for the Lustre programming language. Lustre is a programming language for reactive systems, and its speciality is the focus on value streams. CoCoSpec builds on an assume-guarantee mechanism in the form of mode-dependent and -independent `require` and `ensure` clauses. A mode is a (special) Boolean variables which is specification-only and determines whether a pair of require and ensure clauses are active. To determine the mode, you have to rely on the input or other state variables. There are no explicit modes or transitions between them. Contract automata enforce a stricter computation model, and can easily be encoded into CoCoSpec. NuSCR [20] is an extension of the SCR development and focuses on the system specification with automata. NuSCR automata are similar to contract automata. They have states and edges. The edges hold a trigger condition and assignments. The trigger condition corresponds to the assumption of contracts in contract automata, and the assignments are a more restrictive (and also deterministic) form of guarantees. Note that NuSCR automata are deterministic in general. NuSCR also supports timing constraints in the automata, by specifying an additional condition on edges with a lower and upper time bound. Cimatti et al. [10] provide with OCRA a tool for the design-by-contract of reactive systems. OCRA supports the description of system composition via the same model as our function block diagrams and they also define the refinement relationship between a contract and the inner blocks of a system. OCRA itself is not able to verify program code against a assume-guarantee contract. It focuses only on the refinement relationship. In our work, we adapt their work on the refinement relationship for the use with contract automata as the specification language and additionally close the gap between the contract and the final implementation in the C programming language.

Heizmann et al. [13] use automata in combination with Hoare calculus for the verification of batch programs. In Hoare calculus, the triple $\{pre\}P_1;\ldots;P_n\{post\}$ describes the proof obligation, in which the precondition pre should imply the postcondition $post$ after the execution of the program $P;\ldots;P_n$. For the verification, formulas σ_i are required such that they capture the intermediate states between P_i and P_{i+1}. The authors interpret the flow chart of a program as an automaton A whose alphabet consists of the statements in the program and use the structure of the automaton to infer the required formulas σ_i.

7 CONCLUSION

With contract automata, we presented a specification language for reactive and interactive systems based on the combination of assume-guarantee contracts and finite automata. We support modular verification and composition of systems based on the implementation refinement relationship. This also also allows verification of open programs — programs which can later be extended by new system implementations. We implemented a prototype for the generation of proof obligations that shows the compliance of the software (C program) against a contract automaton, the validity of a composition of multiple systems, and the implementation refinement of a composition against a contract automaton. We used CBMC, SeaHorn, and nuXmv to discharge the proof obligations.

Ongoing and Future Work. Contract automata are the theoretical foundation on which to build a graphical state-machine-based specification language that is comprehensive and understandable for system engineers. While an implementation of this is still underway, we have reported conceptual results [2]. We also plan to support the specification of real-time properties, and support for run-time verification. To make the creation of contract automata easier, we will investigate specification mining as a combination of automata learning techniques and the inference of symbolic relations of input and output variables. To overcome the problem of writing the many required intermediate contracts for each system, a mechanism for the creation of abstract CA for the system interfaces would be beneficial. This could be done by combining the CA of the subsystems or the CA of possible implementations.

ACKNOWLEDGMENTS

This work was supported by funding from the pilot program Core Informatics of the Helmholtz Association (HGF) and from German Federal Ministry for Economic Affairs and Climate Action within the research project SofDCar (19S21002).

REFERENCES

[1] Shaun Azzopardi, Gordon J. Pace, Fernando Schapachnik, and Gerardo Schneider. 2016. Contract automata - An operational view of contracts between interactive parties. *Artif. Intell. Law* 24, 3 (2016), 203–243. https://doi.org/10.1007/S10506-016-9185-2

[2] Joshua Bachmeier, Alexander Weigl, and Bernhard Beckert. 2024. Contract Machines: An Engineer-friendly Specification Language for Mode-Based Systems. In *Methods and Description Languages for Modelling and Verification of Circuits and Systems, MBMV 2024, 27th Workshop, Kaiserslautern, Germany, 14-15 February 2023*. VDE/IEEE. https://doi.org/10.5445/IR/1000165642 (to appear).

[3] Davide Basile and Maurice H. ter Beek. 2023. A Runtime Environment for Contract Automata. In *Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 14000)*, Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker (Eds.). Springer, 550–567. https://doi.org/10.1007/978-3-031-27481-7_31

[4] Bernhard Beckert, Mattias Ulbrich, Birgit Vogel-Heuser, and Alexander Weigl. 2022. 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 (Lecture Notes in Computer Science, Vol. 13572)*, Helmut Seidl, Zhiming Liu, and Corina S. Pasareanu (Eds.). Springer, 7–13. https://doi.org/10.1007/978-3-031-17715-6_2

[5] Marco Bozzano, Alessandro Cimatti, Cristian Mattarei, and Stefano Tonetta. 2014. Formal Safety Assessment via Contract-Based Design. In *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8837)*, Franck Cassez and Jean-François Raskin (Eds.). Springer, 81–97. https://doi.org/10.1007/978-3-319-11936-6_7

[6] Aaron R. Bradley. 2011. SAT-Based Model Checking without Unrolling. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6538)*, Ranjit Jhala and David A. Schmidt (Eds.). Springer, 70–87. https://doi.org/10.1007/978-3-642-18275-4_7

[7] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. 2014. The nuXmv Symbolic Model Checker. In *CAV (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 334–342.

[8] Adrien Champion, Arie Gurfinkel, Temesghen Kahsai, and Cesare Tinelli. 2016. CoCoSpec: A Mode-Aware Contract Language for Reactive Systems. In *Software Engineering and Formal Methods - 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9763)*, Rocco De Nicola and eva Kühn (Eds.). Springer, 347–366. https://doi.org/10.1007/978-3-319-41591-8_24

[9] Chris Chilton, Bengt Jonsson, and Marta Z. Kwiatkowska. 2012. Assume-Guarantee Reasoning for Safe Component Behaviours. In *Formal Aspects of Component Software, 9th International Symposium, FACS 2012, Mountain View, CA, USA, September 12-14, 2012. Revised Selected Papers (Lecture Notes in Computer Science, Vol. 7684)*, Corina S. Pasareanu and Gwen Salaün (Eds.). Springer, 92–109. https://doi.org/10.1007/978-3-642-35861-6_6

[10] Alessandro Cimatti, Michele Dorigatti, and Stefano Tonetta. 2013. OCRA: A tool for checking the refinement of temporal contracts. In *2013 28th IEEE/ACM*

International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013, Ewen Denney, Tevfik Bultan, and Andreas Zeller (Eds.). IEEE, 702–705. https://doi.org/10.1109/ASE.2013.6693137

[11] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 2988)*, Kurt Jensen and Andreas Podelski (Eds.). Springer, 168–176. https://doi.org/10.1007/978-3-540-24730-2_15

[12] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9206)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 343–361. https://doi.org/10.1007/978-3-319-21690-4_20

[13] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2013. Software Model Checking for People Who Love Automata. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 36–52. https://doi.org/10.1007/978-3-642-39799-8_2

[14] Charles Antony Richard Hoare. 1985. *Mathematical logic and programming languages.* Prentice-Hall, Chapter Programs are predicates.

[15] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. 2006. Interface Input/Output Automata. In *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4085)*, Jayadev Misra, Tobias Nipkow, and Emil Sekerinski (Eds.). Springer, 82–97. https://doi.org/10.1007/11813040_7

[16] Barbara Liskov and Jeannette M. Wing. 1994. A Behavioral Notion of Subtyping. *ACM Trans. Program. Lang. Syst.* 16, 6 (1994), 1811–1841. https://doi.org/10.1145/197320.197383

[17] Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. 2021. Toward a Lingua Franca for Deterministic Concurrent Systems. *ACM Trans. Embed. Comput. Syst.* 20, 4, Article 36 (may 2021), 27 pages. https://doi.org/10.1145/3448128

[18] Bertrand Meyer. 1992. Applying "Design by Contract". *Computer* 25, 10 (1992).

[19] Alexander Weigl. 2023. *Companion artifact for Contract Automata: A Specification Language for Mode-Based Systems.* https://doi.org/10.5281/zenodo.10364193

[20] Junbeom Yoo, Tai Hyo Kim, Sung Deok Cha, Jang-Soo Lee, and Han Seong Son. 2005. A formal software requirements specification method for digital nuclear plant protection systems. *J. Syst. Softw.* 74, 1 (2005), 73–83. https://doi.org/10.1016/j.jss.2003.10.018