Contract-Driven Runtime Adaptation

Eunsuk Kang[1(✉)], Akila Ganlath[2], Shatadal Mishra[2], Florin Baiduc[3], and Nejib Ammar[2]

[1] Carnegie Mellon University, Pittsburgh, USA
eskang@cmu.edu
[2] Toyota Motor North America R&D, InfoTech Labs, Ann Arbor, USA
{akila.ganlath,shatadal.mishra,nejib.ammar}@toyota.com
[3] Woven Planet Holdings, Tokyo, Japan
florin.baiduc@woven-planet.global

Abstract. For safe and reliable operation, modern cyber-physical systems (CPS) rely on various assumptions about the environment, such as the reliability of the underlying communication network and the behavior of other, uncontrollable agents. In practice, however, the environment may *deviate* over time, possibly violating one or more of these assumptions. Ideally, in these scenarios, it would be desirable for the system to provide some level of guarantee about its critical requirements. In this paper, we propose a *contract-based* approach to dynamically adapting the behavior of a system component in response to an environmental assumption violation. In particular, we extend the well-known notion of *assume-guarantee contracts* with an additional concept called the *weakening operator*, which describes how the component temporarily weakens its original guarantee to compensate for a violated assumption. Building on this type of contract, which we call an *adaptive contract*, we propose a runtime system for automatically detecting assumption violations and adapting the component behavior. We present a prototype implementation of our adaptation framework on the CARLA simulator and demonstrate its feasibility on an automotive case study.

1 Introduction

To ensure critical requirements, systems rely on various assumptions about their deployment environment. For instance, a driving-assistance system in an autonomous vehicle leverages information about its surroundings (e.g., the distance to and the velocity of the leading vehicle) to maintain a safe distance and minimize the risk of collision. Timely and accurate access to this information, in turn, depends on the performance of the underlying communication network and the reliability of the in-vehicle sensors.

In practice, modern cyber-physical systems (CPS) are deployed in a highly dynamic, uncertain environment where one or more of these assumptions may occasionally fail to hold. An inclement weather, for example, may prevent a sensor from delivering accurate information about the surroundings; a congestion in a traffic area may increase the latency of the network and cause a delay

N. Benz et al. (Eds.): NFM 2024, LNCS 14627, pp. 298–313, 2024.
https://doi.org/10.1007/978-3-031-60698-4_17

in message delivery. Ideally, a robust system would recognize a violation of an assumption and respond appropriately by triggering a fail-safe or fail-operational mechanism (e.g., slow down the vehicle or switch to a manual driver control mode in case of a critical sensor failure).

Contract-based design [1] is a systematic, rigorous methodology for the development of CPS. In this approach, a component is assigned a *contract* that describes its properties or behaviors that are expected by other components (e.g., the client of a service). In this paper, we investigate a type of contract called *assume-guarantee (AG) contract*, where a component is assigned (1) an *assumption*, A, that it makes about its environment (e.g., expected condition over its inputs) and (2) a *guarantee*, G, that it promises to the environment. The concept of AG contracts have been leveraged for a number of different use cases, including verification [23], synthesis [9,13], testing [3], and runtime monitoring [24]. Informally, a well-accepted interpretation of an AG contract [1] is that the component promises to provide the specified guarantee *if* the assumption holds (i.e., $A \Rightarrow G$). One limitation of this interpretation is that an AG contract does not say anything about how the component behaves in case of the assumption violation: Under the standard logical interpretation, $\neg A$ implies anything, meaning that the component may choose to behave in an arbitrary manner and still fulfill its assigned contract.

In this paper, we propose the concept of *adaptive contracts*, which can be used to specify how a component responds to abnormal conditions or changes in the environment, in addition to specifying the expected guarantee under the normative environment. Our intuition is that even if the assumption is violated, the component may still be able to provide some level of guarantee, depending on the degree of the violation. To capture this, our approach extends a standard AG contract with an additional element called the *weakening operator*, which describes the level of *weakened* guarantee that the component promises in case of an assumption violation. Given a component with a contract $C = (A, G)$, suppose that the environment fails to satisfy the original assumption A but satisfies a weaker assumption A' (e.g., the latency of the vehicular communication network increases from 50 to 100 ms). Then, the weakening operator ω maps A' to a corresponding *weakened* guarantee G' that specifies the level of guarantee that the component promises to satisfy under A' (e.g., the vehicle maintains a safe distance at a decreased average velocity, thus temporarily sacrificing its performance in response to the increased network latency).

Building on this notion of an adaptive contract, we propose a runtime mechanism for detecting assumption violations and automatically adapting the behavior of a component in response. In particular, given contract $C = (A, G)$, where A and G are expressed using *Signal Temporal Logic (STL)* [17], we demonstrate how the weakening operator can be specified declaratively as a constraint over the *robustness of satisfaction* [7] of the assumption and guarantee. Furthermore, we show how the task of finding G' for given A' can be formulated as a *mixed-integer linear programming (MILP)* problem.

To demonstrate the feasibility of the proposed approach, we have developed a prototype of the contract-based runtime adaptation mechanism on top of the

CARLA simulator [8]. In particular, we show how our approach can be used to specify a contract for an *adaptive cruise control (ACC)* feature (which relies on assumptions about the reliability of the vehicular network) and enable the component to adjust its guarantee dynamically when these assumptions are violated. Our preliminary experiments show that our adaptation method can be used to achieve an acceptable level of safety during assumption violations.

The paper makes the following contributions:

- The concept of an *adaptive contract*, which extends assume-guarantee contracts with the notion of a *weakening operator* that describes how the component should adjust its guarantee in response to an assumption violation (Sect. 4),
- A runtime adaptation mechanism that leverages MILP to automatically adjust component guarantees (Sect. 5), and
- A prototype implementation of the proposed adaptation mechanism on top of the CARLA simulator and its demonstration on a case study involving ACC (Sect. 6).

2 Motivating Example

As a running example, consider an ACC feature inside a vehicle. When activated, this feature is designed to perform two tasks: (1) continually adjust the acceleration of the ego vehicle to maintain a steady *gap distance* to the leading vehicle and (2) reduce the chance of collision by ensuring that the *time-to-collision (TTC)* is always above some safe threshold (e.g., 3 s). Task (2) is fulfilling a safety requirement, while task (1) is intended to achieve an optimal traffic flow. ACC relies on an internal communication network to receive information about the status of the leading vehicle, such as its relative acceleration, relative velocity, and the relative distance ahead with respect to the ego vehicle.

The system designer assigns an AG contract $C = (A, G)$ to describe the specification of the ACC feature, where A states that the network ensures timely delivery of the information about the leading vehicle (with an upper bound on the message delivery time) and G says that ACC will perform the above two tasks, as long as the assumption A holds. Formally, the contract can also be specified by using a specification language such as STL [17]; e.g., $C = (\varphi_A, \varphi_G)$, where $\varphi_A \equiv \square(\texttt{delay} \leq 100\,\text{ms})$ and $\varphi_G \equiv \square(\lozenge(\texttt{gapDist} \leq 20\,\text{m})) \wedge \square(\texttt{ttc} \geq 3s)$. This contract specification could then be used as part of a runtime monitor to ensure that the environment satisfies its assumption, or that the ACC component fulfills its guarantee as expected.

One limitation of an AG contract, however, is that it does not say anything about what the component should do when its assumption is violated; this is left up to the designer to decide, by devising additional mechanisms to handle those situations. Our approach addresses this by augmenting the AG contract with an additional concept, called the weakening operator, that describes how the component should adapt its behavior in response to an assumption violation. For the ACC feature, this operator specifies how the system should adjust its

guarantee on the TTC and optimal gap distance when the network experiences a delay. In particular, one possible adaptation strategy is to temporarily increase the gap distance to compensate for the network delay, thus weakening part of its original guarantee (i.e., $\square(\lozenge(\texttt{gapDist} \leq 20\,\text{m} + k))$ for some non-negative value k) but still ensuring safety ($\square(\texttt{ttc} \geq 3s)$).

There are important benefits to including this adaptive behavior as part of a component specification. First, this information can be used as part of a runtime adaptive framework that monitors for assumption violation and systematically adjusts the behavior of the component in response. Second, other components in the system may rely on the guarantee provided by C; when its assumption is violated, it may be useful for C to provide some partial guarantee to those components (instead of completely discarding G). Lastly, an adaptive contract encourages the designer to explicitly consider situations in which assumptions are violated and how to respond to them, which may help improve the overall resiliency of the system.

3 Preliminaries

Contract-based Design. *Contract-based design* is a promising methodology for developing CPS [1] by enabling a compositional, "divide and conquer" paradigm where a system is designed as a hierarchy of components, and verification is done on individual components before being composed to provide end-to-end system guarantees.

In the contract framework [1], a *component* M is a basic unit of a system, characterized by a set of *variables* V and a set of *behaviors* (denoted $beh(M)$) expressed over V. Variables are further classified into *input* and *output* variables: Output variables represent those that the component can directly control (e.g., the acceleration of a vehicle), while input variables are observed from the environment but not directly controllable (e.g., the speed of a surrounding vehicle observed through a sensor). Components can be connected to each other through one or more variables and form a larger, composite system.

An *assume-guarantee contract* C is given by pair (A, G), where A and G are *sets* of behaviors for the assumption and guarantee, respectively. Assumptions and guarantees can be expressed in different formalisms, such as temporal logics (e.g., LTL [20], STL [17]) or state machines. Component M *implements* contract C (denoted $M \models C$) if M satisfies its guarantee whenever the assumption holds, i.e., $A \cap beh(M) \subseteq G$. Contract C is *compatible* if there exists a valid environment for M, i.e., if and only if $A \neq \emptyset$, where $\emptyset$ is the empty set. In addition, C is said to be *consistent* if there exists a feasible implementation M for it.

Signal Temporal Logic (STL). In CPS, the behavior of a component can be captured by real-valued *signals*. Formally, a signal s is a function $\mathbf{s} : T \to D$ defined over a finite or infinite set of time, $T \subseteq \mathbb{R}_{\geq 0}$, to a tuple of k real numbers, $D \subseteq \mathbb{R}^k$. Intuitively, the value of signal $\mathbf{s}(t) = (v_1, \ldots, v_k)$ represents the state variables of the system at time t, e.g, v_1 might represent the time-to-collision (TTC) between the ego vehicle and its leading vehicle. For convenience, $\mathbf{s}_i(t)$ denotes the i-th component of the signal at time t (for $1 \leq i \leq k$).

STL [17] is an extension of linear-temporal logic (LTL) [20] that can be used to specify time-varying requirements of a system over real-valued signals. The atomic expression in STL is called a *signal predicate*. A signal predicate μ is a formula of form $f_\mu(\mathbf{s}(t)) \geq 0$, where f_μ is a function from D to $\mathbb{R}$, i.e., the predicate μ is true if and only if $f_\mu(\mathbf{s}(t))$ is at least zero. The overall syntax of an STL formula φ is defined as:

$$\varphi := \mu \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \, \mathcal{U}_{[a,b]} \varphi$$

where $a, b \in \mathbb{R}$, $a < b$, and $\neg$, $\wedge$, and $\mathcal{U}$ are the negation, conjunction, and until operators, respectively. Informally, $\varphi_1 \mathcal{U}_{[a,b]} \varphi_2$ states that φ_1 must hold *until* φ_2 becomes true within time interval $[a, b]$. The operator $\mathcal{U}_{[a,b]}$ can be used to define other temporal operators: $\Diamond_{[a,b]}\varphi := True \, \mathcal{U}_{[a,b]}\varphi$ and $\Box_{[a,b]}\varphi := \neg\Diamond_{[a,b]}\neg\varphi$.

Robustness. Typically, the semantics of temporal logic such as LTL are defined over a *binary* notion of formula satisfaction. STL also supports a *quantitative* notion of satisfaction called *robustness*, which indicates how "close" the system is from satisfying or violating a property. Formally, the robustness of signal $\mathbf{s}$ with respect to formula φ at time t, denoted $\rho(\varphi, \mathbf{s}, t)$, is defined as:

$$\begin{aligned}
\rho(\mu, \mathbf{s}, t) &\equiv f_\mu(\mathbf{s}(t)) \qquad \rho(\neg\varphi, \mathbf{s}, t) \equiv -\rho(\varphi, \mathbf{s}, t) \\
\rho(\varphi_1 \wedge \varphi_2, \mathbf{s}, t) &\equiv \min\{\rho(\varphi_1, \mathbf{s}, t), \rho(\varphi_2, \mathbf{s}, t)\} \\
\rho(\varphi_1 \mathcal{U}_{[a,b]} \varphi_2, \mathbf{s}, t) &\equiv \sup_{t' \in [t+a, t+b]} \min\{\rho(\varphi_2, \mathbf{s}, t'), \inf_{t'' \in [t, t']} \rho(\varphi_1, \mathbf{s}, t'')\} \\
\rho(\Diamond_{[a,b]}\varphi, \mathbf{s}, t) &\equiv \sup_{t_1 \in [t+a, t+b]} \rho(\varphi, \mathbf{s}, t_1) \\
\rho(\Box_{[a,b]}\varphi, \mathbf{s}, t) &\equiv \inf_{t_1 \in [t+a, t+b]} \rho(\varphi, \mathbf{s}, t_1)
\end{aligned}$$

where $\inf_{x \in X} f(x)$ is the greatest lower bound of function $f : X \to \mathbb{R}$ (and sup the least upper bound). The robustness of predicate μ captures the amount by which signal $\mathbf{s}$ at time t is above or below the value of $f_\mu(\mathbf{s}(t))$. For example, consider predicate $\mu \equiv \texttt{ttc}(t) - 3 \geq 0$, which captures the property that "the TTC between the ego and leading vehicles is at least 3.0 s." If, at time t, the TTC signal is $\texttt{ttc}(\mathrm{t}) = 1\,\mathrm{s}$, then robustness value $\rho(\mu, a, t) = -2$ indicates that the system is 2.0 s below the desired safe threshold. In the case of robustness of $\Box_{[a,b]}(\texttt{ttc}(t) - 3 \geq 0)$, it represents the value within interval $[a, b]$ at which the TTC is furthest away from 3.0 s.

4 Adaptive Contracts

While AG contracts are a powerful design and analysis tool, once the component design and corresponding contracts are set, they are sensitive to un-modeled or non-provisioned changes. In particular, an AG contract does not say anything about how the component will behave when perturbations in the environment lead to a violation of the original assumption. These perturbations can be the result of diverse events: hardware failures, frayed wires, unexpected network

congestion, or malicious attacks. In isolation, this can have catastrophic impacts depending on the nature of the component, and in a hierarchical structure, the effects may cascade or propagate to other components and the system as a whole.

An *adaptive contract* extends an assume-guarantee with an additional concept called the *weakening operator* (denoted ω), which describes how the component (implementing the contract) responds to a violation of the assumption by the environment. Formally:

Definition 1 (Adaptive Contract). *An adaptive contract C is tuple (A, G, ω), where A and G are sets of behaviors representing its assumption and guarantee, respectively; ω is a function of type $\mathbb{M}_A \to \mathbb{M}_G$, where $\mathbb{M}_A$ and $\mathbb{M}_G$ are distance metrics for assumptions and guarantees, respectively.*

For a given assumption, A, and the actual behavior manifested by the environment, A' (where A' is weaker than A), let $d_A \in \mathbb{M}_A$ be the distance between A' and A, representing the degree by which the environment violates A. Then, $w(d_A) = d_G \in \mathbb{M}_G$ represents the *maximum degree of weakening* in its guarantee; i.e., the degree by which the original guarantee G may be compromised to a weaker guarantee, G'.

Intuitively, $C = (A, G, \omega)$ is a contract stipulating that any component M implementing C must be designed with a mechanism for providing some level of guarantee even when its assumption is violated. In particular, when A is violated by degree d_A, component M should continue to provide a level of guarantee G' that is no weaker than G by degree $\omega(d_A) = d_G$.

We note a couple of special cases. For some $d_A \in \mathbb{M}_G$ where $d_A \neq 0$, if $\omega(d_A) = 0$ (i.e., no weakening of the guarantee is allowed), any component M implementing C must achieve the original guarantee G even under the assumption violation. If $w(d_A) = \infty$ for some d_A (i.e., the guarantee can be weakened by any arbitrary amount), it means that M needs not provide any guarantee at all when the environment deviates from its assumed behavior by degree d_A. This latter case represents the semantics of a standard AG contract, which leaves unspecified the behavior of the component under an assumption violation.

Realization in STL. The concept of weakening operator ω is general and can be realized through different representations of distance metrics $\mathbb{M}_A$ and $\mathbb{M}_G$. Given our choice of STL as the formalism for specifying A and G, we define ω as a function of type $\mathbb{R} \to \mathbb{R}$, whose domain and range correspond to the robustness of the satisfaction of the assumption and guarantee, respectively. We define the meaning of the weakening operator for STL-based adaptive contracts as follows:

Definition 2 (Robustness-based Weakening Operator). *Given adaptive contract $C = (\varphi_A, \varphi_G, \omega)$ for STL formulas φ_A and φ_G, and weakening operator $\omega : \mathbb{R} \to \mathbb{R}$, $\omega(d_A) = d_G$ for d_A, $d_G \in \mathbb{R}$ if and only if the following holds:*

> *For every signal $\boldsymbol{s}$ and time t, if $\rho(\varphi_A, \boldsymbol{s}, t) \leq 0$ and $-\rho(\varphi_A, \boldsymbol{s}, t) \leq d_A$, then $-\rho(\varphi_G, \boldsymbol{s}, t) \leq d_G$.*

In other words, as long as the environment violates its assumption (φ_A) no more than by d_A, any component implementing C must ensure that its guarantee (φ_G) is violated by no more than d_G.

For example, consider contract $C = (\varphi_A, \varphi_G, \omega)$ for the ACC component, where $\varphi_A \equiv \square(\texttt{delay} \leq 100\,\text{ms})$ and $\varphi_G \equiv \square(\texttt{ttc} \geq 3s) \wedge \square(\lozenge(\texttt{gapDist} \leq 20\,\text{m}))$. Suppose that the condition of the network deteriorates and the assumption about the maximum message delay no longer holds; in particular, let us assume that the new observed delay is $(100 + x)ms$ for some $x > 0$. Thus, for signal $\mathbf{s}$ that represents the observed system execution and current time t, $\rho(\varphi_A, \mathbf{s}, t) = -x$.

To ensure safety (i.e., $\texttt{ttc} \geq 3s$) despite the fact that its information about the leading vehicle may be outdated (by x ms), the ego vehicle must behave more conservatively. To do so, the ACC feature compromises the other part of its guarantee, by maintaining a larger gap distance of $(20 + y)$ meters (for some $y > 0$). With this new behavior, ACC is violating its original guarantee by y; i.e., $\rho(\varphi_G, \mathbf{s}, t) = -y$.

Then, this adaptive behavior of ACC can be represented by $\omega(x) = y$, which informally says that "if the network delay is increased by no more than x ms, then the ACC component will ensure safety with an increase in the gap distance that is no greater than y m".

Specifying the Weakening Operator. As mentioned earlier, the weakening operator ω is a function that maps the degree of an assumption violation (d_A) to the maximum allowed degree of relaxation in the guarantee (d_G). In our approach, ω is specified *symbolically* as a logical constraint R_ω that describes the relationship between variables $d_A, d_G \in \mathbb{R}$ where $w(d_A) = d_G$. Formally:

For every $d_A, d_G \in \mathbb{R}$, $w(d_A) = d_G$ if and only if $R_\omega(d_G, d_A)$ holds true.

Our approach does not prescribe a particular language or type of constraint for specifying R_ω; this will depend on the capability of the underlying reasoning engine to be used as part of the runtime adaptation mechanism. For the particular mechanism that we propose in Sect. 5, we assume that the expressions in R_ω can be encoded as MILP constraints.

Back to our running ACC example, recall that the two parameter values, x and y, correspond to the degrees by which the original assumption and guarantee associated with the contract are violated, respectively. One way to define the relationship between x and y is as follows:

$$R_\omega(x, y) \iff R_1(x, y) \wedge R_2(x, y) \tag{1}$$

$$R_1(x, y) \iff \texttt{timeToCollision(x)} \geq 3 \tag{2}$$

$$R_2(x, y) \iff 20 + y \leq \texttt{estLeadingDist(x)} \tag{3}$$

Here, on line (2), `timeToCollision(x)` is an auxiliary function that estimates the time-to-collision between the leading and ego vehicles, taking into account the additional network delay of x ms. On (3), `estLeadingDist(x)` is a function that computes a conservative estimate of the distance between the leading and ego vehicle, assuming that the leading vehicle continually decelerates

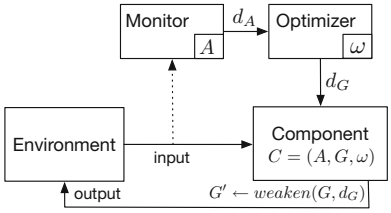

Fig. 1. Conceptual overview of the runtime adaptation framework.

for the time duration x. Due to limited space, we provide the definition of `estLeadingDist(x)` only:

$$\begin{aligned}&\texttt{estLeadingDist(x)}\\&\quad = (\text{leading vehicle displacement}) - (\text{ego vehicle displacement})\\&\quad = (dist_l + v_l * x + 0.5 * \texttt{ACC}_{\texttt{dec}} * x^2) - (v_e * x + 0.5 * a_e * x^2)\end{aligned}$$

where $dist_l$ is the current distance to the leading vehicle, v_l and v_e are the velocities of the leading and ego vehicles, respectively, a_e is the current acceleration of the ego vehicle, and $\texttt{ACC}_{\texttt{dec}}$ is a constant that represents the lowest acceleration value for the leading vehicle. In addition, $dist_l$, v_l, v_e are designated as input variables (i.e., observed from the environment) while a_e is an output variable (i.e., directly controllable by the ACC component). Given that the ego vehicle has possibly outdated information about the acceleration of the leading vehicle (due to the network delay), the gap distance is estimated by assuming the most conservative value for it (i.e., full deceleration, $\texttt{ACC}_{\texttt{dec}}$).

Constraint R_ω captures how the ACC component weakens its guarantee on the gap distance to compensate for the increased network delay (x) by adjusting the acceleration of the ego vehicle. In particular, larger the delay x is, lower the acceleration a_e will be, to keep the time-to-collision above 3 s. This, in turn, results in the vehicle keeping a larger gap distance—thus, a larger value for y.

5 Runtime Adaptation Framework

In this section, we describe an approach for using adaptive contracts as part of a runtime system that detects a violation of an environmental assumption and automatically adjust the behavior of the component in response.

Runtime Architecture. Figure 1 shows the architecture of our proposed runtime adaptation system, consisting of two major parts: (1) a *monitor* and (2) an *optimizer*. Attached to component M with contract $C = (\varphi_A, \varphi_G, \omega)$, the monitor continuously observes the inputs into M and attempts to detect a violation of the assumption by the environment. In particular, the monitor evaluates the robustness of the satisfaction of φ_A over some sequence of input observations $\mathbf{s}_{in}$; if the violation has occurred, it also extracts the robustness value as the degree

of assumption violation, d_A. The optimizer has the knowledge of the weakening operator ω and uses it to compute $d_G = \omega(d_G)$; we describe in the following section how this task is solved using MILP.

Our runtime adaptation method assumes that the behavior of component M can be modified at runtime to dynamically adjust the guarantee that it provides. One way to achieve this is by designing M to expose parameters that can be reconfigured to modify its behavior. For example, an ACC controller may be configured with parameters that represent the expected performance of the vehicle, such as the cruise speed and gap distance [14]; the degree of weakening produced by the optimizer, d_G, could be used to determine the new parameter values. Another way to support dynamic behavior modification is to temporarily override M's behavior with a new sequence of actions to fulfill G'; in the following section, we show how these alternative actions can be synthesized (along with d_G) using MILP.

Finally, when the monitor detects that the environment has resumed satisfying its assumption, the optimizer instructs the component to revert its behavior back to the original one to satisfy G.

MILP-based Adaptation. As stated earlier, when the monitor detects that the environment has violated its assumption by degree d_A, the next step in the adaptation process is to determine how the component should adjust its guarantee; i.e., determine $d_G = \omega(d_A)$. There are different methods to achieving this task, depending on the representation of the operator ω. In this paper, given $\omega(d_A, d_G)$ that is expressed as a symbolic constraint $R_\omega(d_A, d_G)$, we propose a method that formulates this task as a MILP problem.

Our adaptation process produces two outputs: (1) d_G and (2) a sequence of control actions $u_1, ..., u_H \in U$ that the component should execute in order to provide the weakened guarantee. For example, in the ACC example, these actions would correspond to a sequence of acceleration commands that ACC should generate in order to maintain a new gap distance. To produce these outputs, the optimizer carries out a task that is similar to *model predictive control (MPC)* [5], exploring possible sequences of actions over given *prediction horizon* H and evaluating them with respect to the desired property $\varphi_{G'}$. In particular, we adapt the STL-based MPC approach developed by Raman et al. [21], where they also use MILP to generate a sequence of control actions; for details on how to encode STL expressions as MILP, we refer the reader to [21].

Given current system state s_0, prediction horizon $H \in \mathbb{Z}$, predictive model $\mathcal{M}$, and the degree of assumption violation, d_A, the adaptation task is formulated as the following MILP problem:

$$\text{Find } u_1, ..., u_H \in U, d_G \in \mathbb{R} \tag{4}$$

$$\text{that minimizes } d_G \tag{5}$$

$$\text{subject to } \; R(d_A, d_G) \wedge \tag{6}$$

$$\mathbf{s}' = predict(\mathcal{M}, s_0, [u_1, ..., u_H]) \wedge \tag{7}$$

$$-\rho(\varphi_G, \mathbf{s}', 0) \leq d_G \tag{8}$$

Expressions on (6)–(8) are constraints that must be satisfied by the solution to the variables $u_1, ..., u_H$ and d_G. On (6), $R(d_A, d_G)$ is the constraint that relates the degree of assumption violation with that of guarantee weakening (as described in Sect. 4). Function *predict* on (7) takes model $\mathcal{M}$ and produces a predictive signal, $\mathbf{s}'$, that describes how the system might evolve over the horizon H given a possible sequence of actions $u_1, ..., u_H$. Then, the constraint on (8) states that over this predicted execution, the system must provide a level of guarantee that is no weaker than by d_G.

A predictive model ($\mathcal{M}$) takes the current state of the system s_0 and control action $u \in U$, and produces the next state. Inside *predict*, this model is executed repeatedly over $u_1, ..., u_H$ to produce the predictive signal $\mathbf{s}'$. This model can be specified as a transition system, a dynamical model, or a lookup table [25] that maps each (state, action) pair to the corresponding next state.

Finally, (5) stipulates that the degree of weakening d_G should be minimized. This optimization objective is to ensure that the original guarantee is weakened no more than needed to satisfy the constraints; minimizing this is desirable since in general, weaker guarantees mean a lower or degraded level of functionality.

Example. Recall contract $C = (\varphi_A, \varphi_G, \omega)$ for the ACC component, where $\varphi_A \equiv \square(\texttt{delay} \leq 100\,\text{ms})$ and $\varphi_G \equiv \square(\texttt{ttc} \geq 3s) \wedge \square(\lozenge(\texttt{gapDist} \leq 20\,\text{m}))$. The predictive model $\mathcal{M}$ is specified as a dynamical model that describes how the velocity of the ego vehicle and the distance to the leading vehicle evolve based on a given command for setting the ego acceleration (i.e., the control action u). When the monitor detects that the network delay exceeds the original threshold (e.g., 500 ms), it computes the degree of violation d_A as (500–100) = 400 ms. The optimizer then takes this value and generates the corresponding MILP problem for computing d_G, where $R(d_A, d_G)$ encodes the constraints (1)–(3) from Sect. 4. Finally, the underlying MILP solver computes $d_G = 2$, suggesting that the behavior of the ACC component should be adjusted to maintain a larger gap of 22 m instead of the original 20 m to achieve the safe TTC threshold of 3 s.

6 Case Study

We have developed a prototype implementation of the proposed runtime adaptation framework and applied it to a case study involving a realistic implementation of ACC. We take the component M_{acc} to be the composition of (1) the ACC functions `estLeadingDist`, `timeToCollision` and (2) a proportional-integral-derivative (PID) controller to track the safe relative distance set-point $\texttt{gapDist}_{\texttt{des}}$. The component takes input signals from a ranging sensor (e.g., RADAR), which provides the current relative distance and velocity, and produces an acceleration command generated by the PID controller.

The design goal for M_{acc} is to fulfill the *safety* and *performance* requirements, similar to those described in Sect. 2. In particular, the adaptive contract assigned to M_{acc} is as follows: $C = (\varphi_A, \varphi_G, \omega)$, where $\varphi_A \equiv \square(\texttt{delay} \leq T)$ and $\varphi_G \equiv \square(\texttt{gapDist} \geq 5m) \wedge \square(\lozenge(\texttt{gapDist} \leq 20\,\text{m}))$. Note that for simplicity, instead of

Fig. 2. Test scenario for evaluation of the ACC case study

`ttc` from the running example, we use `gapDist` to specify the safety requirement. We assume that the PID controller has been tuned to enforce these requirements based on expected ranges of `delay`. In our experiments, we allow this network latency to vary outside the design-stage ranges, resulting in a violation of φ_A and motivating the need for contract adaptation.

Recall that the contract guarantee G can be broken into two parts: $\varphi_G \equiv \varphi_{G_{safety}} \wedge \varphi_{G_{perf}}$. Hence, weakening either one of the two requirements would amount to weakening the overall component guarantee. As satisfying the safety requirement are paramount to automotive engineers, we weaken $\varphi_{G_{perf}}$ in favor of $\varphi_{G_{safety}}$ by allowing the ACC component to temporarily increase its tracking distance in response to an increased network delay: $\texttt{gapDist}_{\texttt{des}} \leftarrow \texttt{gapDist}_{\texttt{des}} + \texttt{d}_{\texttt{G}}$; that is, the *runtime adaptation* for the ACC component is realized by adjusting the setpoint of the PID controller.

Implementation in CARLA. The component M_{acc} is implemented as a node in ROS2 [16] and deployed in CARLA [8], a high-fidelity simulator for autonomous vehicles, which serves as the environment for M_{acc}. The component obtains information about the vehicle and environment through simulated sensors and generates throttle commands. The parameters (e.g., `gapDist`) for the ACC component are configurable during runtime. The sensor network delay is implemented by buffering messages before they are delivered to M_{acc}.

In accordance with Fig. 1, the Monitor is realized as a ROS2 node wrapping the Runtime Assurance Monitoring Tool (RTAMT) [19], a software library for automatic generation and deployment of monitors for STL specifications. In addition, the Optimizer is implemented as a ROS2 node wrapping MiniZinc, an open-source constrained optimization toolchain that provides a high-level modeling language for encoding and solving MILP constraints [18].

Experiments. We developed a car following scenario in CARLA as the test environment for our experiments. In this scenario, the leading vehicle starts from rest and tracks a predefined path of length 244 m with a constant speed set-point of 35 m/s. This is enforced by the default PID controllers provided by CARLA for both the lateral and longitudinal dynamics. The ego vehicle starts from rest at a fixed distance behind the leading vehicle, as can be seen in Fig. 2. The lateral control is managed by the default PID controller while the longitudinal control

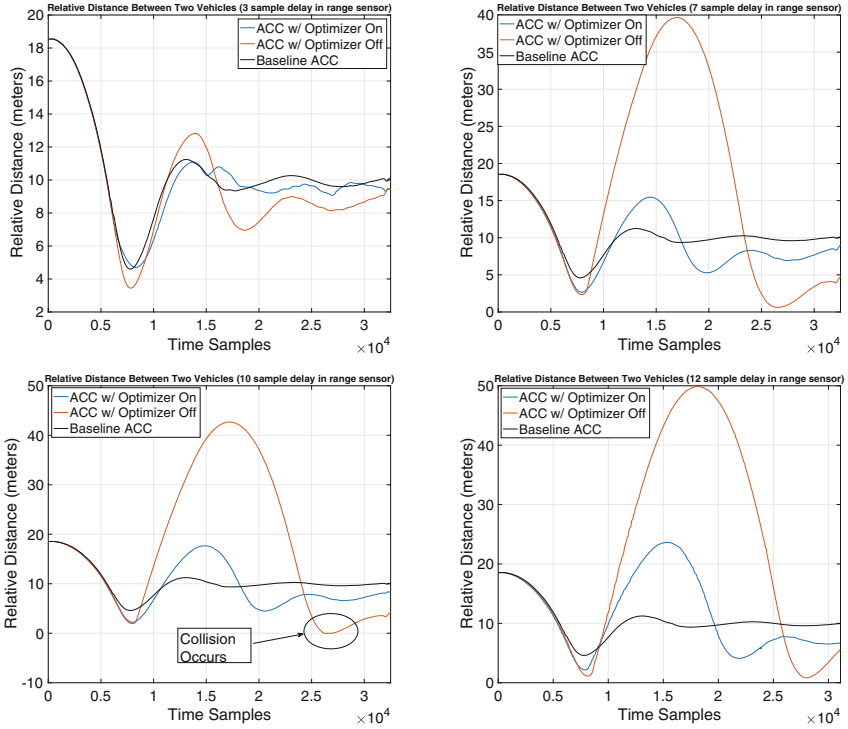

Fig. 3. Relative distance between two vehicles. The controller with and without contract adaptation is shown in red and blue, respectively; the black plot represents the baseline controller behavior without time delay (i.e., adaptation is never triggered). From top left, in a clockwise fashion, the delay samples are 3, 7, 10 and 12. (Color figure online)

is provided by M_{acc} component, which is activated at the start of the scenario. The scenario terminates when the leading vehicle reaches a preset destination at the end of the path.

Experiments were performed by simulating the above scenario under varying amounts of network delay in the range sensor samples; for each `delay`, the scenario was simulated twice, with and without the adaptation enabled (i.e., without the adaption, the ACC component would try to maintain its original set-point, ignoring the delay). Delay was added to the range sensor by buffering the signal by 3, 7, 10 and 12 samples, respectively, for each experiment.

Results. Figure 3 shows the performance of the proposed approach, where the relative distance between the leading and ego vehicles is plotted for the different scenarios. The system performance without contract adaptation is shown in red lines in the plots whereas the performance of the contract adaptation framework is shown in blue. As a reference, the baseline relative distance is shown in black, for where there is no network delay and no contract adaptation (baseline ACC).

As the delay in the range sensor is increased, the relative distance between the two vehicles grows significantly when there is no contract adaptation. We believe that this occurs because the delay in the range sensor effectively delays the response at which the controller generates control signals, reducing its performance compared to the baseline component. On the other hand, the contract adaptation strategy mitigates the delay by increasing the relative tracking distance, which correspondingly increases the gap distance and provides the adapted component with a larger response time margin. As the network delay is increased, this trend continues, resulting in the un-adapted system experiencing a collision and near-collision for `delay` values 7 and 10, respectively. Though the performance of the system with contract adaptation also degraded with increasing network delay, it was able to avoid such critical failures. In summary, these results show that the contract-based adaptation approach is viable and has a potential to improve the system resiliency against environmental deviations.

We also measured the runtime overhead of the adaptation process, in terms of the amount of time it took for the solver to generate the weakening parameter d_G. In general, we found that the overhead was acceptable (around 80 ms on average) and did not interfere with the system operation, although it is possible that for a more complex application, the overhead could be larger.

Despite this potential benefit from contract adaptation, we also note that as the ranging sensor latency `delay` increased, the number of failed Optimizer calls (e.g. no feasible solution to problem (4)–(8)) also increased. In these situations, the implementation M_{acc} used the baseline `gapDist`, to default to the behavior of the system without adaptation. In addition, there was significant variance in the solver computation time when such a solution existed. As part of future work, we plan to further investigate this phenomenon, including the impact of different solvers, solver configurations, and alternative MILP formulations.

7 Related Work

The problem of responding to faulty or unexpected environmental behaviors is not new. One closely related concept is that of *graceful degradation* [10], which refers to mechanisms for maintaining system functionality at a reduced level in presence of an unexpected component or environmental failure. Although this is a well-studied topic, there is relatively little prior work on formal specification of graceful degradation (with the work by Merlihy and Wing [12] being a notable exception but for distributed systems). An adaptive contract can be regarded as a component specification that explicitly indicates the level of degraded functionality that the component can provide under abnormal environmental conditions.

The concept of the weakening operator was in part inspired by the concept of *stability* in control theory [15], which stipulates that bounded disturbances in a system input should result in bounded disturbances in the output. Within formal methods, researchers have proposed definitions of *robustness* (not to be confused with the robustness of satisfaction in STL) that capture a similar concept [2,11, 26]. For example, Bloem et al. propose a notion of robustness that relates the

number of incorrect environment inputs and system outputs [2]; Tabuada et al. propose a different notion of robustness that assigns costs to certain inputs and outputs (e.g., a high cost may be assigned to an input that deviates significantly from the expected behavior) and states that an input with a small cost should only result in an output with a proportionally small cost [26]. As far as we know, these works mainly focus on design-stage verification or synthesis rather than leveraging these notions of robustness for monitoring and behavior adaptation.

Self-adaptive systems refer to systems that are capable of dynamically adjusting their behaviors in response to changes in the environment [27]. Among the prior works in this area, closest to our work are those that leverage temporal logics to specify the system requirements to be achieved during adaptation [4,6,22,28]. Requirement relaxation (or weakening) has also been investigated in the context of self-adaptation; most notably, RELAX [28] is a framework based on fuzzy logic that supports specifications of requirements that explicitly capture uncertainty about possible system behavior. RELAX can be used to support self-adaptation mechanisms where the system dynamically adjusts its behavior to accommodate for uncertainty or changes in the environment. One interesting future direction that we plan to explore is to leverage RELAX as another type of requirements specification language (instead of STL) to support contract-based adaptation. The work by Chu et al. [6] proposes a method for weakening component guarantees that is also based on STL, although their goal is to dynamically resolve *feature interactions* (i.e., unexpected conflicts among components), rather than weakening a guarantee in response to an environmental deviation.

8 Limitations and Future Work

We have proposed the notion of an *adaptive contract*, which explicitly captures how a component behaves in response to the violation of an assumption by the environment. In addition, we have presented an approach for dynamically adapting the behavior of the component by leveraging an adaptive contract specified in STL and an encoding of the adaptation task in MILP, demonstrating its feasibility through a car-following case study in the automotive domain.

Although these results show a promise, further research is needed to overcome a number of challenges to make this type of adaptation approach effective under realistic settings. First, the process of solving MILP problems can be computationally expensive and introduce a variable amount of runtime overhead. We plan to explore alternative methods that leverage domain-specific heuristics or offline learning to enable more efficient adaptation. Second, our approach currently assumes that the system is able to satisfy the weakened (as well as the original) guarantees. In practice, however, it is possible for these guarantees to be violated due to a fault within the system; thus, augmenting the type of runtime architecture that we have presented with monitoring of guarantees (in addition to assumptions) is an interesting extension that would provide an additional layer of assurance.

Furthermore, we have only studied the application of an adaptive contract to a single component (e.g., ACC); an interesting follow-up work could investigate how the *composition* of multiple adaptive contracts can enable reasoning about the end-to-end robustness of a system against environmental disturbances. Finally, under certain "good" environmental conditions (e.g., higher than expected network performance), it may be possible to apply a similar type of adaptation to *strengthen* the guarantee (instead of weakening it), to provide an optimal level of system functionality by enabling dynamically flexible component specifications. We plan to investigate a hybrid adaptation framework that is capable of both weakening and strengthening component guarantees.

References

1. Benveniste, A., et al.: Contracts for system design. Found. Trends Electron. Des. Autom. **12**(2–3), 124–400 (2018)
2. Bloem, R., Chatterjee, K., Greimel, K., Henzinger, T.A., Jobstmann, B.: Specification-centered robustness. In: International Symposium on Industrial Embedded Systems (SIES) (2011)
3. Blundell, C., Giannakopoulou, D., Pasareanu, C.S.: Assume-guarantee testing. ACM SIGSOFT Softw. Eng. Notes **31**(2) (2006)
4. Calinescu, R., Grunske, L., Kwiatkowska, M.Z., Mirandola, R., Tamburrelli, G.: Dynamic QoS management and optimization in service-based systems. IEEE Trans. Softw. Eng. **37**(3), 387–409 (2011)
5. Camacho, E., Alba, C.: Model Predictive Control. Advanced Textbooks in Control and Signal Processing. Springer, London (2013). https://doi.org/10.1007/978-3-319-24853-0
6. Chu, S., et al.: Runtime resolution of feature interactions through adaptive requirement weakening. In: IEEE/ACM Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), pp. 115–125. IEEE (2023)
7. Donzé, A., Maler, O.: Robust satisfaction of temporal logic over real-valued signals. In: FORMATS (2010)
8. Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., Koltun, V.: CARLA: an open urban driving simulator. In: Annual Conference on Robot Learning (2017)
9. Ghasemi, K., Sadraddini, S., Belta, C.: Compositional synthesis via a convex parameterization of assume-guarantee contracts. In: HSCC (2020)
10. González, O., Shrikumar, H., Stankovic, J.A., Ramamritham, K.: Adaptive fault tolerance and graceful degradation under dynamic hard real-time scheduling. In: RTSS (1997)
11. Henzinger, T.A., Otop, J., Samanta, R.: Lipschitz robustness of finite-state transducers. In: FSTTCS (2014)
12. Herlihy, M., Wing, J.M.: Specifying graceful degradation. IEEE Trans. Parallel Distrib. Syst. **2**(1), 93–104 (1991)
13. Iannopollo, A., Tripakis, S., Sangiovanni-Vincentelli, A.L.: Specification decomposition for synthesis from libraries of LTL assume/guarantee contracts. In: DATE, pp. 1574–1579 (2018)
14. Kesting, A., Treiber, M., Schönhof, M., Helbing, D.: Adaptive cruise control design for active congestion avoidance. Transp. Res. Part C: Emerg. Technol. **16**(6), 668–683 (2008)

15. Kirk, D.: Optimal Control Theory: An Introduction. Dover Books on Electrical Engineering Series. Dover Publications, New York (2004)
16. Macenski, S., Foote, T., Gerkey, B., Lalancette, C., Woodall, W.: Robot operating system 2: design, architecture, and uses in the wild. Sci. Robot. **7**(66) (2022)
17. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: FORMATS, pp. 152–166 (2004)
18. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: Minizinc: towards a standard CP modelling language. In: International Conference on Principles and Practice of Constraint Programming (CP), pp. 529–543 (2007)
19. Nickovic, D., Yamaguchi, T.: RTAMT: Online robustness monitors from STL (2020). https://arxiv.org/abs/2005.11827
20. Pnueli, A.: The temporal logic of programs. In: Annual Symposium on Foundations of Computer Science (FOCS), pp. 46–57 (1977)
21. Raman, V., Donzé, A., Maasoumy, M., Murray, R.M., Sangiovanni-Vincentelli, A.L., Seshia, S.A.: Model predictive control with signal temporal logic specifications. In: IEEE Conference on Decision and Control (CDC) (2014)
22. Rodrigues, A., Caldas, R.D., Rodrigues, G.N., Vogel, T., Pelliccione, P.: A learning approach to enhance assurances for real-time self-adaptive systems. In: SEAMS (2018)
23. Saoud, A., Girard, A., Fribourg, L.: On the composition of discrete and continuous-time assume-guarantee contracts for invariance. In: European Control Conference, ECC, pp. 435–440 (2018)
24. Sokolsky, O., Zhang, T., Lee, I., McDougall, M.: Monitoring assumptions in assume-guarantee contracts. In: PrePost@IFM (2016)
25. Sundström, C., Frisk, E., Nielsen, L.: Diagnostic method combining the lookup tables and fault models applied on a hybrid electric vehicle. IEEE Trans. Control Syst. Technol. **24**(3), 1109–1117 (2016)
26. Tabuada, P., Balkan, A., Caliskan, S.Y., Shoukry, Y., Majumdar, R.: Input-output robustness for discrete systems. In: EMSOFT, pp. 217–226 (2012)
27. Weyns, D.: An Introduction to Self-adaptive Systems: A Contemporary Software Engineering Perspective. Wiley-IEEE Computer Society, Hoboken (2020)
28. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H.C., Bruel, J.: RELAX: a language to address uncertainty in self-adaptive systems requirement. Requir. Eng. **15**(2), 177–196 (2010)