Contract-Based Specification Refinement and Repair for Mission Planning

Piergiuseppe Mallozzi[1], Inigo Incer[1], Pierluigi Nuzzo[2], Alberto Sangiovanni-Vincentelli[1]
[1]Department of Electrical Engineering and Computer Sciences, UC Berkeley, USA
[2]Ming Hsieh Department of Electrical and Computer Engineering, University of Southern California, USA
Email: {mallozzi,inigo,alberto}@berkeley.edu, nuzzo@usc.edu

***Abstract*—We address the problem of modeling, refining, and repairing formal specifications for robotic missions using assume-guarantee contracts. We show how to model mission specifications at various levels of abstraction and implement them using a library of pre-implemented specifications. Suppose the specification cannot be met using components from the library. In that case, we compute a proxy for the best approximation to the specification that can be generated using elements from the library. Afterward, we propose a systematic way to either 1) *search* for and refine the 'missing part' of the specification that the library cannot meet or 2) *repair* the current specification such that the existing library can refine it. Our methodology for searching and repairing mission requirements leverages the quotient, separation, composition, and merging operations between contracts.**

I. Introduction

Mission *specification* is a mission formulation in a formal (e.g., logical) language with precise semantics [2]. Many results in the literature highlight the advantages of specifying robotic missions in a temporal logic language, like linear temporal logic (LTL) or computation tree logic (CTL) [14], [16], [19], [27]–[29], [32], [44]. Producing a suitable *implementation* of a mission specification is the problem of finding a policy to be followed by the robot such that the mission specification is always satisfied. For specifications in LTL, reactive synthesis can automatically generate correct-by-construction implementations from a given specification [10], [13], [16], [22], [26].

Contract-based modeling [4], [11], [34], [36]–[38], [43] can be suited to formalize and analyze properties of reactive systems. A *contract* specifies the behavior of a component by distinguishing its responsibilities (*guarantees*) from those of its environment (*assumptions*). It is possible to use contracts to model the mission specification and automatically realize its implementation using reactive synthesis [25]. However, reactive synthesis is impractical for large specifications due to its high computational complexity (double exponential in the length of the formula, in the general case). Breaking the specification into more manageable parts that can be realized independently can reduce computational complexity. Moreover, instead of directly generating the implementation, we can search for efficient implementations that can be composed and together realize the specification. It is then essential to decouple the specification of a generic robotic mission from its possible implementations. Different implementations can, for example, refer to different robotic systems on which the mission will be deployed.

System specifications formalized with contracts can be incrementally refined using a *library* of pre-defined components [20], [24], [45]. In robotic missions, a component is a pre-implemented mission specification. A library containing such components can be used to find suitable refinements to the mission specification. Using a library, we can also *adapt* the specification to support a variety of possible implementations. For example, a general 'search and rescue' mission can be automatically adapted to be deployed in different environments (e.g., different map configurations). Ideally, the refinement process should use elements in the library of pre-implemented specifications. Since every component of the library can be pre-implemented, we would not need to generate implementations for every specification, but we can *reuse* existing ones. However, various libraries can model different robotic systems or system aspects, and each library can add additional constraints that the specification may need to account for. Yet, the designer might not be aware *a priori* of the additional constraints posed by library or the library itself may not be rich enough to "cover" the specification. The question is then how to add the minimum number of components to the library to refine the specification completely. Alternatively, we may ask how to minimally revise the specification to enable an effective implementation with the given library.

In this paper, we present CR^2 (Contract-based specification Refinement and Repair), a structured methodology to model, search and repair formal specifications represented as assume-guarantee contracts [4]. The search process consists in keeping the specification fixed while searching for the *missing part*, e.g., in other libraries. The repair process consists of automatically *patching* the current specification so the available library can refine it. Whenever a specification cannot be refined from the library, we propose an algorithm to produce the best 'candidate selection' of elements such that the 'missing part' to search or repair is minimal, thus maximizing utilization of the available library.

This research was supported in part by the Wallenberg AI Autonomous Systems and Software Program (WASP), funded by the Knut and Alice Wallenberg Foundation, by the National Science Foundation (NSF) under Awards 1846524 and 2139982, the Office of Naval Research (ONR) under Award N00014-20-1-2258, the Defense Advanced Research Projects Agency (DARPA) under Award HR00112010003, and the Okawa Foundation.

The contributions of this paper are the following:

- A framework to model mission specifications and prove specification refinements across abstraction layers.
- An algorithm to find a candidate selection of library elements implementing a mission specification that is optimal with respect to a 'specification coverage' metric based on syntactic and semantic similarity notions.
- A methodology to search or repair specifications that cannot be refined. The search is based on the application of the contract operations of *quotient* [21] and *composition* [4] while the repair is based on the operations of *separation* and *merging* [42].

We have implemented CR[2] in a tool that supports the designer in specification modeling, refinement, and repair[1].

Related Work: Repair of system specifications is a widely studied problem in the literature. Recent approaches have focused on repairing LTL specifications that are unrealizable for reactive synthesis [1], [7], [8], [23], [30]. These approaches focus on the discovery of *new assumptions* to make the specification realizable. They target a restricted fragment of LTL specifications (e.g., GR(1)) and mostly use model checking techniques. A more recent approach by Gaaloul et al. [17] targets signal-based modeling notations, rather than LTL specifications, and relies on testing, rather than model checking, to generate the data that are used to learn assumptions via machine learning.

Brizio et al. [5] use a search-based approach to repair LTL specifications. Their approach is based on syntactic and semantic similarity as well as *model counting*, i.e., the number of models that satisfy the formula. The new realizable specification is then produced by successive applications of genetic operations. For repairing Signal Temporal Logic (STL) formulas, Gosh et al. [18] propose algorithms to detect possible reasons for infeasibility and suggest repairs to make it realizable. Other approaches [9], [12], instead of repairing the specification, focus on repairing the *system* in a way that it can satisfy the specification. In the robotics domain, Boteanu et al. [40] focus on adding assumptions to the robot specification while having the human prompted to confirm or reject them. Pacheck et al. [40], [41] automatically encode into LTL formulas robot capabilities based on sensor data. If a task cannot be performed, they suggest skills that enable the robot to complete the task.

Our framework uses LTL specifications, but instead of mining for new assumptions or changing the system (i.e., the pre-implemented library elements), it repairs the existing specifications based on what can be refined from the library. Our repairs are always the smallest (in terms of the set of behaviors removed from the original specification) and completely automated, since they are based on algebraic operations. Finally, we use notions of semantic and syntactic similarity [5] to compute the 'closest' candidate compositions of library elements to the specification to be refined.

[1]Tool available at https://rebrand.ly/refine-repair-tool.

II. Background

We provide some background on assume-guarantee contracts and linear temporal logic.

A. Assume-Guarantee Contracts

Contract-based design has emerged as a design paradigm capable of providing formal support for building complex systems in a modular way by enabling compositional reasoning, step-wise refinement of specifications, and reuse of pre-designed components [4], [35], [37], [39].

A *contract* $\mathcal{C}$ is a triple (V, A, G) where V is a set of system *variables* (including, e.g., input and output variables or ports), and A and G—the assumptions and guarantees—are sets of behaviors over V. For simplicity, whenever possible, we drop V from the definition and refer to contracts as pairs of assumptions and guarantees, i.e., $\mathcal{C} = (A, G)$. A expresses the behaviors expected from the environment, while G expresses the behaviors that an implementation promises under the environment assumptions. In this paper, we express assumptions and guarantees as sets of behaviors satisfying a logical formula; we then use the formula itself to denote them. An environment E satisfies a contract $\mathcal{C}$ whenever E and $\mathcal{C}$ are defined over the same set of variables, and all the behaviors of E are included in the assumptions of $\mathcal{C}$, i.e., when $|E| \subseteq A$, where $|E|$ is the set of behaviors of E. An implementation M satisfies a contract $\mathcal{C}$ whenever M and $\mathcal{C}$ are defined over the same set of variables, and all the behaviors of M are included in the guarantees of $\mathcal{C}$ when considered in the context of the assumptions A, i.e., when $|M| \cap A \subseteq G$.

A contract $\mathcal{C} = (A, G)$ can be placed in saturated form by re-defining its guarantees as $G_{sat} = G \cup \overline{A}$, where $\overline{A}$ denotes the complement of A. A contract and its saturated form are semantically equivalent, i.e., they have the same set of environments and implementations. Therefore, in the rest of the paper, we assume that all the contracts are expressed in saturated form. In particular, the relations and operations we will discuss are only defined for contracts in saturated form. A contract $\mathcal{C}$ is *compatible* if there exists an environment for it, i.e., if and only if $A \neq \emptyset$. Similarly, a saturated contract $\mathcal{C}$ is *consistent* if and only if there exists an implementation satisfying it, i.e., if and only if $G \neq \emptyset$. We say that a contract is *well-formed* if and only if it is compatible and consistent. We detail below the contract operations and relations used in this paper.

a) Refinement: Refinement establishes a pre-order between contracts, which formalizes the notion of replacement. Let $\mathcal{C} = (A, G)$ and $\mathcal{C}' = (A', G')$ be two contracts, we say that $\mathcal{C}$ refines $\mathcal{C}'$, denoted by $\mathcal{C} \preceq \mathcal{C}'$, if and only if all the assumptions of $\mathcal{C}'$ are contained in the assumptions of $\mathcal{C}$ and all the guarantees of $\mathcal{C}$ are included in the guarantees of $\mathcal{C}'$, that is, if and only if $A \supseteq A'$ and $G \subseteq G'$. Refinement entails relaxing the assumptions and strengthening the guarantees. When $\mathcal{C} \preceq \mathcal{C}'$, we also say that $\mathcal{C}'$ is an *abstraction* of $\mathcal{C}$ and can be replaced by $\mathcal{C}$ in the design.

b) Composition: The operation of composition ($\|$) is used to generate the specification of a system made of components that adhere to the contracts being composed. Let $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$ be two contracts. The composition $\mathcal{C} = (A, G) = \mathcal{C}_1 \parallel \mathcal{C}_2$ can be computed as follows:

$$A = (A_1 \cap A_2) \cup \overline{(G_1 \cap G_2)}, \quad (1)$$

$$G = G_1 \cap G_2. \quad (2)$$

Intuitively, an implementation satisfying $\mathcal{C}$ must satisfy the guarantees of both $\mathcal{C}_1$ and $\mathcal{C}_2$, hence the operation of intersection in (2). An environment for $\mathcal{C}$ should also satisfy all the assumptions, motivating the conjunction of A_1 and A_2 in (1). However, part of the assumptions in $\mathcal{C}_1$ may be already supported by $\mathcal{C}_2$ and *vice versa*. This allows relaxing $A_1 \cap A_2$ with the complement of the guarantees of $\mathcal{C}$ [4].

c) Quotient (or Residual): Given two contracts $\mathcal{C}_1$ and $\mathcal{C}'$, the quotient $\mathcal{C}_2 = (A_2, G_2) = \mathcal{C}'/\mathcal{C}_1$ is defined as the largest specification (with respect to the refinement order) that we can compose with $\mathcal{C}_1$ so that the result refines $\mathcal{C}'$. In other words, the quotient is used to find the specifications of missing components. We can compute the quotient [21] as follows:

$$A_2 = A' \cap G_1 \quad \text{and} \quad G_2 = G' \cap A_1 \cup \overline{(A' \cap G_1)}.$$

d) Merging: Contracts that handle specifications of various viewpoints of the same design element can be combined using the *merging* operation [42]. Given $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$ their *merger* contract, denoted $\mathcal{C} = \mathcal{C}_1 \bullet \mathcal{C}_2$, is the contract which promises the guarantees of both specifications when the assumptions of both specifications are respected, that is,

$$\mathcal{C} = (A_1 \cap A_2, G_1 \cap G_2 \cup \overline{A_1 \cap A_2}).$$

e) Separation: Given two contracts $\mathcal{C}_1$ and $\mathcal{C}'$, the operation of *separation* [42] computes the contract $\mathcal{C}_2 = (A_2, G_2) = \mathcal{C}' \div \mathcal{C}_1$ as

$$A_2 = A' \cap G_1 \cup \overline{(G' \cap A_1)} \quad \text{and} \quad G_2 = G' \cap A_1.$$

The contract $\mathcal{C}_2$ is defined as the smallest (with respect to the refinement order) contract satisfying $\mathcal{C}' \preceq \mathcal{C}_1 \bullet \mathcal{C}_2$.

B. Linear Temporal Logic

Given a set of atomic propositions AP (i.e., Boolean statements over system variables) and the state s of a system (i.e., a specific valuation of the system variables), we say that s *satisfies* p, written $s \models p$, with $p \in AP$, if p is *true* at state s. We can construct LTL formulas over AP according to the following recursive grammar:

$$\varphi := p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \mathbf{X}\ \varphi \mid \varphi_1\ \mathbf{U}\ \varphi_2$$

where φ, φ_1, and φ_2 are LTL formulas. From the negation ($\neg$) and disjunction ($\vee$) of the formula we can define the conjunction ($\wedge$), implication ($\rightarrow$), and equivalence ($\leftrightarrow$). Boolean constants *true* and *false* are defined as $\mathit{true} = \varphi \vee \neg\varphi$ and $\mathit{false} = \neg\mathit{true}$. The temporal operator **X** stands for *next* and **U** for *until*. Other temporal operators such as *globally* (**G**) and *eventually* (**F**) can be derived as follows: $\mathbf{F}\ \varphi = \mathit{true}\ \mathbf{U}\ \varphi$ and $\mathbf{G}\ \varphi = \neg(\mathbf{F}(\neg\varphi))$. We refer to the literature [3] for the formal semantics of LTL. For the rest of the paper, we indicate with $\overline{\varphi}$ the negation of φ, i.e., $\neg\varphi$.

An LTL formula can be *realized* by a controller via reactive synthesis [15]. Reactive synthesis generates a controller $\mathcal{M}$ (a finite state machine) from a specification φ (an LTL formula), where the atomic propositions are partitioned into inputs and outputs. If a controller can be synthesized, it is guaranteed to satisfy the specification under all possible inputs. If such a machine exists, we say that $\mathcal{M}$ *realizes* φ.

III. Problem Definition

Robotic missions state what the robot should achieve in the world. We model each robot objective with a *contract*.

Definition III.1 (Mission Specification)**.** A mission specification is a contract $\mathcal{C} = (\varphi_A, \varphi_G)$ consisting of a pair of LTL formulas.

A behavior is an infinite sequence of states, where each state is an assignment of values to all system variables within their domain. A finite state machine is a tuple $\mathcal{M} = (S, \mathcal{I}, \mathcal{O}, s_0, \delta)$ where S is the set of states, $s_0 \in S$ is the initial state, and $\delta : S \times 2^{\mathcal{I}} \rightarrow S \times 2^{\mathcal{O}}$ is the transition function.

Definition III.2 (Library of Components)**.** A library of components is a pair $\Delta = (K, T)$, where $K = \{\mathcal{L}'_1, \mathcal{L}'_2, \ldots, \mathcal{L}'_n\}$ is a set of n contracts and $T = \{\mathcal{M}'_1, \mathcal{M}'_2, \ldots, \mathcal{M}'_n\}$ is a set of n finite state machines satisfying the corresponding contracts, i.e., such that $\mathcal{M}'_i \models \mathcal{L}'_i$ for all i.

The library of components bridges the gap between a general specification and a specific set of implementations that can be executed in a certain environment. The robot (controller) can be modeled as a finite state machine, obtained by composition of library components, which satisfies the mission specification.

Definition III.3 (Mission Satisfaction Problem)**.** Given a mission specification $\mathcal{C}$ and a library of components $\Delta = (K, T)$, produce an implementation $\mathcal{M} = \mathcal{M}_1 \parallel \mathcal{M}_2 \parallel \ldots \parallel \mathcal{M}_p$ where $\{\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_p\} \in T$ such that $\mathcal{M} \models \mathcal{C}$.

However, we cannot always find components in the library that can satisfy the mission specification, e.g., the library may not 'cover' all the requirements of the mission. When there exists no $\mathcal{M}$ in the library such that $\mathcal{M} \models \mathcal{C}$, we propose two strategies: *1)* loosening the specification $\mathcal{C}$ by relaxing its constraints or *2)* extending the library with new components that can accommodate the requirements in $\mathcal{C}$. Our framework, named CR^2, automatically performs both strategies by leveraging the contract algebra while satisfying certain optimality criteria.

IV. Running Example

Let us consider an environment modeling a general store formed by a front, a back, and an entrance. The left part of Figure 1 shows such a map with three locations: L_F (*front*),

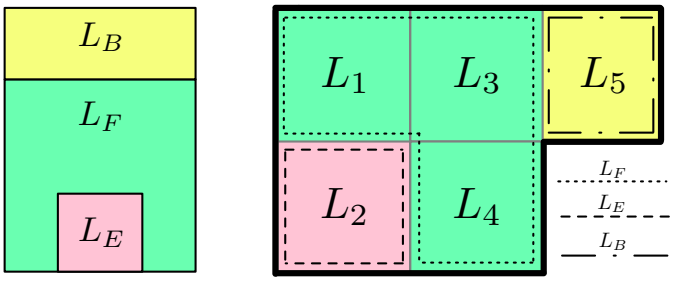

Fig. 1: Location maps used for the running example. The map on the left is 'refined' by the map on the right.

L_B (*back*), and L_E (*entrance*). Location L_E is inside L_F. We assume the robot is equipped with a sensor to detect people and actuators to greet them. The mission requires moving between the back and the front of the store and greeting customers when they are detected.

We want to deploy the mission on a specific store with the map shown on the right side of Figure 1. We have five locations, $L_1, \ldots, L_5$, which indicate where the robot can be. Locations L_1, L_3, and L_4 represent the front of the store, location L_2 the entrance, and location L_5 the back. Let us assume that a library of components Δ contains implementations of mission objectives that the robot can perform in the more detailed map.

Our goal is to *refine* a general mission specification with a set of implementations specific to the particular environment. In the rest of the paper, we will see examples of how CR^2 tackles this problem, even when the refinement is not immediately possible.

V. Modeling Framework

We introduce the building blocks of the modeling infrastructure used in CR^2, starting with the concept of *types*. Types are used to assign *semantics* to every location, sensor, and action in the mission. Relationships among types are used to automatically generate constraints to model the world in which the robot operates. We call *world context* the ground constraints that model the world.

Definition V.1 (Type)**.** A type is a semantic concept related to the mission (e.g., a location, an action, or a sensor) that is used to generate the world context. We indicate with Θ the set of all types in scope.

We indicate types with capital letters and atomic propositions with lower-case letters. If not explicitly specified, we assume that an atomic proposition p belongs to a type denoted by the corresponding capital letter P.

Types can be related to other types in four ways: via mutual exclusion, adjacency, extension (or subtyping), and covering.

Definition V.2 (Mutual Exclusion)**.** A type $A \in \Theta$ is mutually exclusive from a type $B \in \Theta$ if instances of A and B can never be *true* simultaneously.

This relationship will be used to state that the robot cannot be in two locations simultaneously.

Definition V.3 (Adjacency)**.** A type $A \in \Theta$ is adjacent to a type $B \in \Theta$ if instances of B can become *true* one step after instances of A are *true*.

This relationship will help us specify that the reachable locations from a given location in one timestep are those allowed by the current map.

Definition V.4 (Extension)**.** Let $A \in \Theta$ and $B \in \Theta$ be two types, where $A \neq B$. We say that A is a *subtype* of B or that A *extends* B, denoted by $A \preceq B$, if the concept A is included in the concept B.

Subtyping is used to relate abstract types to concrete types. For example, in Figure 1, we will define a type L_F denoting "the robot is in the front of the store" and another denoting "the robot is in location L_4." Because L_4 is part of the front, we will say that L_4 is a subtype of L_F.

Definition V.5 (Covering)**.** Let $A \in \Theta$ be a type and $A'_i \in \Theta$, $1 \leq i \leq n$, be subtypes of A. We say that the set $\{A'_1, \ldots, A'_n\}$ covers the type A if, when an atomic proposition of A is true, the atomic propositions of at least one of the A'_i are true.

The concept of covering allow us to say that an abstract type is represented exactly by a disjunction of concrete types. For example, in Figure 1, the type "the robot is in the front" is covered by the set $\{L_1, L_3, L_4\}$ since being in the front requires the robot to be in one of those specific locations.

Definition V.6 (Similar Types)**.** A type $A \in \Theta$ is *similar* to a type $B \in \Theta$ if and only if (iff) $A \preceq B$ or $A = B$.

Our modeling framework uses types to generate world context constraints semantically. For each type of relationship described above, our framework produces an LTL formula that can be added to the world context.

Example V.1. *We consider the mission specification of our running example to be the following contract $\mathcal{C}$:*

$$\mathcal{C} \begin{cases} A = & \mathsf{InfOften}(s) \\ G = & \mathsf{OrderedPatrolling}(l_b, l_f) \wedge \\ & \wedge\ \mathsf{InstantaneousReaction}(s, g), \end{cases} \tag{3}$$

where s and g are the atomic propositions indicating that the robot is sensing a person and greeting, respectively, and l_b, l_f are atomic propositions indicating the location of the robot. InfOften *(infinitely often) represents the LTL construct used to express* 'globally eventually.' OrderedPatrolling *is a robotic pattern, i.e., a template used to conveniently capture a robotic specification, that expresses the continuous visit of a set of locations with a precise order.* InstantaneousReaction *is another robotic pattern which, in the same time step, requires performing an action, i.e., setting g to true, based on the truth value of s. We refer to the literature [31] for more details on the robotic patterns used in this paper.*

VI. Generating the Context Constraints

We show how CR^2 generates the context constraints and verifies *well-formedness*, *refinement*, and *realizability* of spec-

ifications. In our modeling framework, types are used to generate world context constraints. For each kind of relationship described in Section V our framework produces an LTL formula which can be added to the world context.

Given a formula φ, we use AP^{φ} to denote the set of atomic propositions that appear in φ; we call the *types of* φ the set of types associated with the atomic propositions that appear in φ. For example, if $\varphi = l_1$, $AP^{\varphi} = \{l_1\}$, and the types of φ consist of the set $\{L_1\}$. We define the following functions:

- $MTX(\varphi)$ produces an LTL formula enforcing the mutual exclusion conditions for the types of φ. For each pair of atomic propositions $p_i, p_j \in AP^{\varphi}$ whose types P_i and P_j, respectively, are mutually exclusive, we append the constraint $\mathsf{G}(p_i \rightarrow \overline{p_j}) \wedge \mathsf{G}(p_j \rightarrow \overline{p_i})$, i.e.,

$$MTX(\varphi) = \bigwedge_{\substack{p_i, p_j \in AP^{\varphi} \\ P_i \textit{mutex with} P_j}} \mathsf{G}(p_i \rightarrow \overline{p_j}) \wedge \mathsf{G}(p_j \rightarrow \overline{p_i}).$$

- $ADJ(\varphi)$ produces an LTL formula enforcing the adjacency conditions of all adjacent types of φ. For each pair $p_i, p_j \in AP^{\varphi}$ whose corresponding types P_i and P_j are related via adjacency, we include the constraint $\mathsf{G}(p_i \rightarrow \mathsf{X}(p_i \vee p_j)) \wedge \mathsf{G}(p_j \rightarrow \mathsf{X}(p_j \vee p_i))$, i.e.,

$$ADJ(\varphi) = \bigwedge_{\substack{p_i, p_j \in AP^{\varphi} \\ P_i \textit{adj. with} P_j}} \mathsf{G}(p_i \rightarrow \mathsf{X}(p_i \vee p_j)) \wedge \mathsf{G}(p_j \rightarrow \mathsf{X}(p_j \vee p_i)).$$

- EXT produces an LTL formula enforcing all the extension relations. For each pair p_i, p_j such that $P_i \preceq P_j$, EXT includes the formula $\mathsf{G}(p_i \rightarrow p_j)$, i.e.,

$$EXT = \bigwedge_{P_i \preceq P_j} \mathsf{G}(p_i \rightarrow p_j).$$

- COV produces an LTL formula enforcing the coverage constraints among all types that satisfy such constraints. For atomic propositions p_a, p_b, where $b \in \mathcal{I}$ holds, $\mathcal{I}$ being a set of indexes, and such that $\{P_b\}_{b \in \mathcal{I}}$ covers P_a, we include the constraint $\mathsf{G}(p_a \rightarrow \bigvee_{b \in \mathcal{I}} p_b)$, that is,

$$COV = \bigwedge_{\{P_b\}_{b \in \mathcal{I}} cov. P_a} \mathsf{G}\left(p_a \rightarrow \bigvee_{b \in \mathcal{I}} p_b\right).$$

Example VI.1. *In our running example, we have two maps at two levels of abstraction. There is an 'abstract map' with locations L_B, L_F, and L_E and a 'concrete map' with locations L_1, L_2, L_3, L_4, and L_5.*

We assign a type to every location on the map, whose instances are atomic propositions, e.g., L_5 has an associated atomic proposition l_5. Whenever l_5 is true, then L_5 is also true, and the robot is in location L_5 on the map. We also define the type S to model a sensor that detects the presence of a person and the type G to model the greeting action. We use s and g for the atomic propositions corresponding to types S and G, respectively.

For every type we define its mutual exclusion, adjacency, extension, and covering relationships. For example, L_1 has an adjacency relationship with types L_2 and L_3. It is mutually exclusive with L_2, L_3, L_4, and L_5, since the robot cannot be in multiple locations at the same time. L_1 extends L_F, i.e., $L_1 \preceq L_F$ and L_1 is part of the $\{L_1, L_3, L_4\}$ covering of L_F.

VII. Contract-Based Verification

Once the designer uses CR^2 to define the types as discussed above, many relationships between atomic propositions can be automatically inferred, according to the expressions for MXT, ADJ, EXT, and COV. CR^2 can then check for well-formedness, refinement, and realizability. CR^2 performs well-formedness checks on the mission specification as well as the components in the library. Refinement checks are performed to verify whether a specification is more 'stringent' than another. This is particularly important when checking whether a composition of elements from the library can meet a top-level specification.

a) Well-Formedness Check: For any contract having φ_A and φ_G as assumptions and guarantees, we check that both φ_A and φ_G are satisfiable by checking the satisfiability of the following formulas:

$$\varphi_A \wedge MTX(\varphi_A) \wedge ADJ(\varphi_A),$$
$$\varphi_G \wedge MTX(\varphi_G) \wedge ADJ(\varphi_G),$$

where we only consider MTX and ADJ as context constraints, since we are only concerned with the satisfiability of the formulas for a single contract. For example, let l_b and l_f be atomic propositions associated with locations L_B and L_F. If the designer formulates a specification having as guarantees $l_b \wedge l_f$ and *true* as assumptions, CR^2 finds that $l_b \wedge l_f \wedge \mathsf{G}(l_b \rightarrow \overline{l_f}) \wedge \mathsf{G}(l_f \rightarrow \overline{l_b}) \wedge \mathsf{G}(l_b \rightarrow \mathsf{X}(l_b \vee l_f)) \wedge \mathsf{G}(l_f \rightarrow \mathsf{X}(l_f \vee l_b))$ is unsatisfiable, hence the contract is not well-formed. For simplicity, we did not include the adjacency relationships for all types.

b) Refinement Check: In refinement verification, we need to take into consideration the context constraints given by EXT and COV because these connect abstract types with concrete subtypes and coverings. Let $\mathcal{C}_1 = (\varphi_{A1}, \varphi_{G1})$ and $\mathcal{C}_2 = (\varphi_{A2}, \varphi_{G2})$ be two contracts. To prove that $\mathcal{C}_1 \preceq \mathcal{C}_2$, we have to check whether $\varphi_{G1} \rightarrow \varphi_{G2}$ and $\varphi_{A2} \rightarrow \varphi_{A1}$ are *valid* formulas. When checking for the validity of a formula $\phi = \varphi_1 \rightarrow \varphi_2$, CR^2 first checks the *satisfiability* of the formulas:

$$\varphi_1 \wedge MTX(\varphi_1) \wedge ADJ(\varphi_1), \tag{4}$$
$$\varphi_2 \wedge MTX(\varphi_2) \wedge ADJ(\varphi_2). \tag{5}$$

If they are satisfiable, it proceeds to verify the *validity* of the implication ϕ in the world context:

$$(EXT \wedge COV) \rightarrow \phi. \tag{6}$$

Example VII.1. *Suppose that we want to check whether, by* Patrolling *locations L_1 and L_3 in the concrete map, the robot is also* Patrolling *location L_F in the abstract map.* Patrolling

is a robotic pattern [31] that requires the robot to visit a location infinitely often. We want to prove that $(\mathsf{GF}(l_1) \wedge \mathsf{GF}(l_3))$ *implies* $\mathsf{GF}(l_f)$*. After checking the satisfiability of the formulas* (4) *and* (5)*, CR*^{2} *proves the validity of the formula*

$$\mathsf{G}(l_1 \rightarrow l_f) \wedge \mathsf{G}(l_3 \rightarrow l_f) \rightarrow (\mathsf{GF}l_1 \wedge \mathsf{GF}l_3 \rightarrow \mathsf{GF}l_f). \quad (7)$$

Since (7) *is valid, we can conclude that a robot, by visiting the locations* L_1 *and* L_3 *infinitely often, is also visiting location* L_F *infinitely often.*

Remark. The formula (7) is a simplified version of (6). We do not always need the context to enforce coverage and extensions among *all* the types of a formula. In this example, it is sufficient to include the extension relationships among l_1, l_3, and l_f to prove the refinement. However, if the formula ϕ in (6) is *not* monotone, meaning that some of the atomic propositions appear negated and some not, then it is necessary to add the constraints generated by COV to the context constraints.

c) Realizability Check: We say that a contract $\mathcal{C} = (\varphi_A, \varphi_G)$ is *realizable* if there exists a finite state machine that implements the formula

$$\phi = MTX(\varphi_A) \wedge ADJ(\varphi_A) \wedge \varphi_A \rightarrow \quad (8)$$
$$MTX(\varphi_G) \wedge ADJ(\varphi_G) \wedge \varphi_G \quad (9)$$

via reactive synthesis. Since the synthesis process only concerns a single contract, the context constraints generated by EXT and COV are unnecessary. For example, to implement the Patrolling of locations l_1 and l_3 in the previous example, CR^2 checks the realizability of the following formula:

$$\mathsf{G}(l_1 \rightarrow \overline{l_3}) \wedge \mathsf{G}(l_3 \rightarrow \overline{l_1}) \wedge$$
$$\wedge \mathsf{G}(l_1 \rightarrow \mathsf{X}(l_1 \vee l_2 \vee l_3)) \wedge \mathsf{G}(l_3 \rightarrow \mathsf{X}(l_3 \vee l_1 \vee l_4 \vee l_5)) \wedge$$
$$\wedge \mathsf{GF}(l_1) \wedge \mathsf{GF}(l_3).$$

Our framework automatically checks the well-formedness of every contract and all the refinement relationships among them. These checks are translated into model checking problems, and NuSMV [6] is used to solve them. On the other hand, we use STRIX [33] to check the realizability of the contracts in the library, when their implementation is not available and to produce Mealy machines that implement them when they are realizable.

VIII. Contract Search and Repair

Figure 2 shows the processes involved in the refinement of a contract $\mathcal{C}$ using a library of components Δ. First, CR^2 searches for the best *candidate composition* of contracts $\hat{\mathcal{L}}$ from Δ, i.e., the selection of contracts that once composed maximize a heuristic function related to the refinement of $\mathcal{C}$. Then, according to the result of a *refinement analysis* procedure, CR^2 can either declare the refinement *complete*, start a *search* procedure, start a *repair* procedure, or declare the refinement *failed*. In the search process we look for new contracts to be able to refine $\mathcal{C}$, whereas in the repair process we automatically modify $\mathcal{C}$ such that Δ can refine it. In the following sections, we discuss *1)* the selection of an optimal candidate composition, *2)* the refinement analysis procedure, *3)* the search process using the contract operations of quotient and composition, *4)* the repair process using the contract operations of separation and merging.

A. Finding the Best Candidate Composition

The best candidate composition $\hat{\mathcal{L}}$ is the composition of a set of contracts in the library that aims to be *'the closest refinement'* of $\mathcal{C}$ that can be generated from Δ. We define the closest refinement by formulating a heuristic function h based on *1)* type similarity, and *2)* behavior coverage. Given $\mathcal{C}$ and Δ, the best candidate composition is the composition of contracts in Δ that maximizes h.

Let Γ be the class of all contracts and Θ the set of all types. We define the following functions:

Definition VIII.1 (Similarity Score)**.** $SIM : \Gamma \times 2^{\Theta} \rightarrow \mathbb{N}$ is a function that takes as input a contract $\mathcal{C} \in \Gamma$ and a set of types $\Theta_i \in 2^{\Theta}$ and returns the number of *similar types* (as in Definition V.6) between $\mathcal{C}$ and Θ_i. Let $\Gamma_i \in 2^{\Gamma}$ be a set of contracts in Γ. For each Γ_i, let Θ_i denote the set of all the types of the contracts in Γ_i, and let N be the number of types of a contract $\mathcal{C}$ in Γ. The *similarity score* of a set of contracts Γ_i with respect to $\mathcal{C}$ is

$$\frac{SIM(\mathcal{C}, \Theta_i)}{N} \times 100, \quad (10)$$

that is, the percentage of similar types 'covered' by Γ_i.

Definition VIII.2 (Refinement Score)**.** The *refinement score* of contract $\mathcal{C}$ with respect to a set of contracts $\Gamma_i \in 2^{\Gamma}$ is a function $REF : \Gamma \times 2^{\Gamma} \rightarrow \mathbb{R}$ that returns the percentage of contracts in Γ_i that can be refined by $\mathcal{C}$.

To compute the refinement score of each contract in the library with respect to the entire library, a pair-wise comparison can be made among all the possible selections of contracts in the library. This process can be performed independently from the search for the best candidate composition.

If no composition in the library is a valid refinement of the top level contract, our framework searches for the best candidate composition by looking at the possible combinations of contracts that are composable and computing for each of them their similarity score with respect to $\mathcal{C}$. Let Ω be the set of compositions with the highest similarity score. If Ω has more than one element, i.e., if $|\Omega| > 1$, then CR^2 keeps in Ω only the compositions generated by the *least number of contracts*. Then, if $|\Omega| > 1$, we compute the refinement score of the compositions in Ω. The best candidate composition is the element in Ω having the highest refinement score. We choose one randomly if more than one element has the highest score.

Example VIII.1. *Let us consider a simplified version* $\mathcal{C}_1$ *of* $\mathcal{C}$ *in (3), which only contains* $\mathsf{OrderedPatrolling}(l_f, l_b)$ *in the*

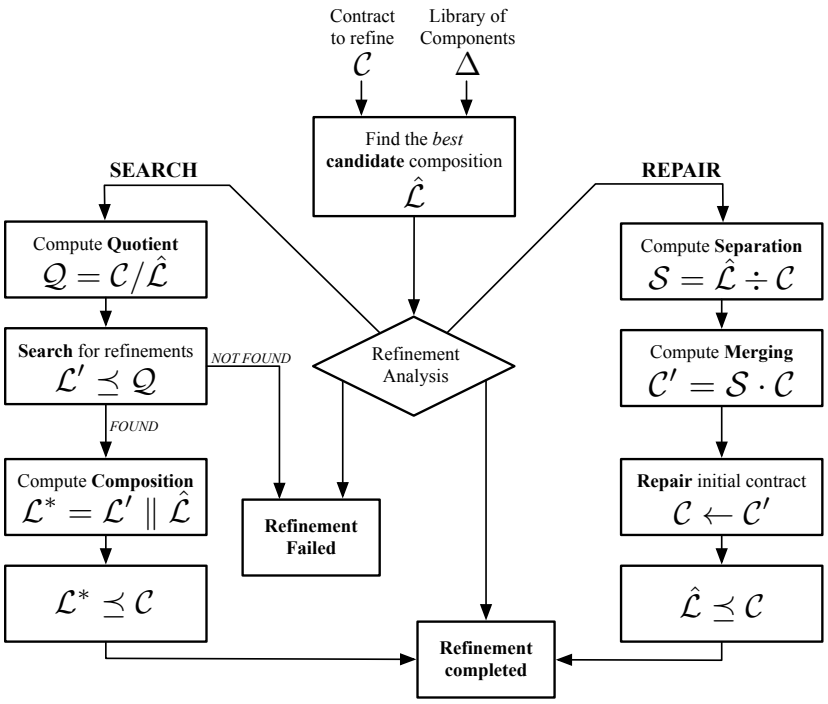

Fig. 2: Flow diagram showing all the processes involved in creating a refinement from a library.

guarantees:

$$\mathcal{C}_1 \begin{cases} A = & \textit{true} \\ G = & \mathsf{GF}(l_f \wedge \mathsf{F}l_b) \wedge (\overline{l_b}\ \mathsf{U}\ l_f) \wedge \\ & \wedge\ \mathsf{G}(l_b \rightarrow \mathsf{X}(\overline{l_b}\ \mathsf{U}\ l_f)) \wedge \mathsf{G}(l_f \rightarrow \mathsf{X}(\overline{l_f}\ \mathsf{U}\ l_b)). \end{cases} \tag{11}$$

Contract (11) requires to repeatedly visit locations l_f and l_b, i.e., the associated atomic propositions must be infinitely often true. Furthermore, it imposes that the locations be visited in a fixed order, starting from l_f.

Let us assume that our library of contracts $\Delta = (K, M)$ has $K = \{\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, \mathcal{L}_4\}$ defined as

$$\mathcal{L}_1 \begin{cases} G = & \mathsf{Patrolling}(l_5) \\ & = \mathsf{GF}l_5 \end{cases} \quad \mathcal{L}_2 \begin{cases} G = & \mathsf{Patrolling}(l_3) \\ & = \mathsf{GF}l_3 \end{cases}$$

$$\mathcal{L}_3 \begin{cases} G = & \mathsf{Visit}(l_3, l_1) \\ & = \mathsf{F}l_3 \wedge \mathsf{F}l_1 \end{cases} \quad \mathcal{L}_4 \begin{cases} G = & \mathsf{Visit}(l_5) \\ & = \mathsf{F}l_5 \end{cases}$$

When not differently stated, we consider the assumptions to be true. For the guarantees of each contract above we indicate both the robotic pattern and its LTL representation.

When computing the composition of contracts in K that has the best similarity and refinement scores, seven candidates have a similarity score of 100% between $\mathcal{C}_1$ and all the types in Δ . Among these, three result from the composition of two contracts, while the rest are composed of more than three contracts. After filtering out the candidates formed by the composition of more than two contracts, CR^2 chooses the candidate with the highest refinement score, which, in this case, is $\hat{\mathcal{L}} = \mathcal{L}_1 \parallel \mathcal{L}_2$, with the following guarantees:

$$\mathsf{GF}l_5 \wedge \mathsf{GF}l_3. \tag{12}$$

Intuitively, we found the composition of contracts in Δ that generates behaviors that are most 'similar' to those of contract (11). Therefore, we can maximize the set of behaviors of $\mathcal{C}_1$ that can be covered by Δ.

B. Refinement Analysis

The refinement analysis evaluates the best candidate composition $\hat{\mathcal{L}}$ and determines the appropriate strategy to complete the refinement of $\mathcal{C}$ from Δ. The outcome of the analysis can be one of the following:

- *Refinement Failed*: $\mathcal{C}$ can not be refined by Δ.
- *Refinement Completed*: $\hat{\mathcal{L}}$ is already a refinement of $\mathcal{C}$.
- *Start Search Procedure*: $\hat{\mathcal{L}} \not\preceq \Delta$, start the search procedure for a new specification using contract quotient and composition.
- *Start Repair Procedure*: $\hat{\mathcal{L}} \not\preceq \Delta$, start a repair procedure to modify $\hat{\mathcal{C}}$ such that Δ can refine it. This process uses contract merging and separation.

Algorithm 1 shows the main steps of the refinement analysis procedure. In addition to $\mathcal{C}$ and Δ, the algorithm can take as input additional libraries $D = \{\Delta', \Delta'', \dots\}$. Moreover, users can express their intention of performing a *repair* or a *search* procedure. If designers do not express their preference, CR^2 chooses a procedure based on the similarity score. If the library *covers* all the types of $\mathcal{C}$ or its similarity score is at least 80%, the algorithm performs a repair of the specification. Otherwise, other types can likely be found in the additional libraries. If different libraries are available and the similarity score is less than 80%, CR^2 performs the search procedure. The following sections describe the search and repair procedures and illustrate how they are applied to our running example.

Algorithm 1: Refinement Analysis

Input: $\mathcal{C}$: contract to refine, Δ: library of components, $\hat{\mathcal{L}}$: best candidate composition, $D = \{\Delta', \Delta'', \dots\}$: additional libraries of components (*optional*), `repair`, `search`: Boolean arguments indicating the designer preference (*optional*)

Output: `refinement_complete`: Boolean indicating that the refinement procedure has been completed, `refinement`: contract refining $\mathcal{C}$

if $\hat{\mathcal{L}} \preceq \Delta$ **then**
  `/* ` $\hat{\mathcal{L}}$ `is already a refinement of` Δ `*/`
  return *true*, $\hat{\mathcal{L}}$
if *similarity score* $== 0\%$ **then**
  `/* Refinement failed */`
  return *false*, *None*
if *repair* **then**
  `/* Designer is 'forcing' a repair */`
  return repair_procedure$(\hat{\mathcal{L}}, \mathcal{C})$
if *search* **then**
  `/* Designer is 'forcing' a search */`
  return search_procedure$(\hat{\mathcal{L}}, \mathcal{C}, D)$
`/* Choose Search or Repair based on the similarity score */`
if *similarity score* $\geq 80\%$ **then**
  return repair_procedure$(\hat{\mathcal{L}}, \mathcal{C})$
else
  return search_procedure$(\hat{\mathcal{L}}, \mathcal{C}, D)$

C. Specification Search via Quotient and Composition

The best candidate composition $\hat{\mathcal{L}}$ in Section VIII-A tends to be the most refined composition of contracts, *delegating* as much functionality as possible to the library of contracts Δ. We then need to find the specification that Δ cannot meet but we still need to satisfy in order to refine $\mathcal{C}$.

To find this missing part given $\hat{\mathcal{L}}$ and $\mathcal{C}$, we would like to have a specification that is as general as possible. We then resort to the operation of quotient, as it produces the most abstract specification that, composed with $\hat{\mathcal{L}}$, can refine $\mathcal{C}$. *Any refinement* of the quotient can be substituted in the composition, and we still obtain a refinement of $\mathcal{C}$.

As shown in Figure 2, we first compute the *quotient* between $\mathcal{C}$ and $\hat{\mathcal{L}}$, i.e., $\mathcal{Q} = \mathcal{C}/\hat{\mathcal{L}}$. Then, we refine the quotient by searching for a new specification $\mathcal{L}'$ such that $\mathcal{L}' \preceq \mathcal{Q}$. The refinement $\mathcal{L}'$ can be searched in a library of contracts $\Delta' \in D$. The *search_procedure* will search the libraries in D, and once a refinement of the quotient is found, it is composed with $\hat{\mathcal{L}}$, i.e., $\mathcal{L}^* = \mathcal{L} \parallel \hat{\mathcal{L}}_i$. The resulting contract $\mathcal{L}^*$ is guaranteed to refine $\mathcal{C}$. If there is no refinement of the quotient in any of the libraries in D, then the refinement process fails. At this point, the designer could choose to *delegate* to some third party the implementation of the quotient by giving them $\mathcal{Q}$.

Example VIII.2. *Let us continue with the example in the previous section, where we found that the best candidate composition of $\mathcal{C}_1$ in (11) using the library Δ is $\hat{\mathcal{L}} = \mathcal{L}_1 \parallel \mathcal{L}_2$. Even though the similarity score is maximum, let us search for new contracts in the new library Δ'. We compute the quotient $\mathcal{Q} = \mathcal{C}_1/\hat{\mathcal{L}}$:*

$$\mathcal{Q} = \begin{cases} A = & \mathsf{GF} l_5 \wedge \mathsf{GF} l_3 \\ G = & (\mathsf{GF}(l_f \wedge \mathsf{F} l_b) \wedge (\overline{l_b}\ \mathsf{U}\ l_f) \quad \wedge \\ & \wedge \quad \mathsf{G}(l_b \rightarrow \mathsf{X}(\overline{l_b}\ \mathsf{U}\ l_f)) \wedge \mathsf{G}(l_f \rightarrow \mathsf{X}(\overline{l_f}\ \mathsf{U}\ l_b))) \vee \\ & \vee \quad \overline{\mathsf{GF} l_5 \wedge \mathsf{GF} l_3}. \end{cases}$$

The quotient is the result of an algebraic expression and is computed automatically; without looking at the specifications, the designer knows the missing behavior from library Δ such that $\mathcal{C}$ can be refined. In fact, any refinement of $\mathcal{Q}$ can 'complete' the candidate composition $\hat{\mathcal{L}}$ so that it refines $\mathcal{C}$. CR^2 searches for refinements of $\mathcal{Q}$ in Δ' to produce a new candidate composition. Let $\mathcal{L}'$ in (13) be the candidate composition that completely refines $\mathcal{Q}$. $\mathcal{L}'$ indicates a strict order among locations to be visited, similar to the StrictOrderedPatrolling robotic patterns, but without prescribing that they be continuously visited.

$$\mathcal{L}' \begin{cases} A = & \mathit{true} \\ G = & (\overline{l_5}\ \mathsf{U}\ l_3) \wedge \quad \mathsf{G}(l_5 \rightarrow \mathsf{X}(\overline{l_5}\ \mathsf{U}\ l_3)) \quad \wedge \\ & \wedge \quad \mathsf{G}(l_3 \rightarrow \mathsf{X}(\overline{l_3}\ \mathsf{U}\ l_5)). \end{cases} \tag{13}$$

In contrast to (11), the strict order among locations, i.e., $l_3 \rightarrow l_5$, found in (13) does not allow locations l_3 or l_5 to be visited more than once per round of visits. We have that $\mathcal{L}' \preceq \mathcal{Q}$. However, neither $\hat{\mathcal{L}}$ nor $\mathcal{L}'$ refine $\mathcal{C}_1$. Contract (13) only imposes a strict visit order, but does not impose to actually visit the locations. To complete the refinement, we produce a new contract $\mathcal{L}^* = \hat{\mathcal{L}} \parallel \mathcal{L}'$ which now refines $\mathcal{C}_1$.

D. Specification Repair via Separation and Merging

Instead of finding the missing element from Δ, the *repair* operation attempts to make a minimal modification of the top-level specification $\mathcal{C}$ so that the library Δ can implement it. Let $\hat{\mathcal{L}}$ be the candidate composition from library Δ, as before. As shown in Figure 2, we first compute the *separation* between $\hat{\mathcal{L}}$ and $\mathcal{C}$, i.e., $\mathcal{S} = \hat{\mathcal{L}} \div \mathcal{C}$. We then *merge* it with $\mathcal{C}$, generating a new contract $\mathcal{C}'$. Now we can repair $\mathcal{C}$ by *replacing* it with $\mathcal{C}'$, which is guaranteed to be refined by the candidate composition $\hat{\mathcal{L}}$. In contrast to the quotient, the operation of separation, combined with merging, will give us the *smallest abstraction* of $\hat{\mathcal{L}}$ such that $\mathcal{C}$ merged with $\mathcal{S}$ can refine it [42].

Example VIII.3. *Let us consider contract $\mathcal{C}_2$, another simplified version of $\mathcal{C}$, which only prescribes that the robot always greet immediately when a person is detected (i.e., in the same time-step), assuming that people will always eventually be detected. We obtain:*

$$\mathcal{C}_2 \begin{cases} A = & \mathsf{GF}(s) \\ G = & \mathsf{GF}(s) \rightarrow \mathsf{G}(s \rightarrow g). \end{cases}$$

Let us assume that in our library the candidate composition $\hat{\mathcal{L}}$ is the following contract:

$$\hat{\mathcal{L}} \begin{cases} A = & \mathit{true} \\ G = & \mathsf{G}(s \rightarrow \mathsf{X} g). \end{cases}$$

$\hat{\mathcal{L}}$ requires the robot to greet the person in the next time instant, once a person is detected. Obviously, $\hat{\mathcal{L}}$ fails to refine $\mathcal{C}_2$. Let us now compute the separation between $\hat{\mathcal{L}}$ and $\mathcal{C}_2$, obtaining the following contract:

$$\mathcal{S}\begin{cases} A = & \mathsf{G}(s \rightarrow g) \vee \overline{(\mathsf{G}(s \rightarrow \mathsf{X}g) \wedge \mathsf{GF}s)} \\ G = & \mathsf{G}(s \rightarrow \mathsf{X}g) \wedge \mathsf{GF}s. \end{cases}$$

The result of $\mathcal{S}$ merged with $\mathcal{C}_2$ is the following contract:

$$\mathcal{C}_2'\begin{cases} A = & \mathsf{GF}s \wedge (\mathsf{G}(s \rightarrow g) \vee \overline{(\mathsf{GF}s \wedge \mathsf{G}(s \rightarrow \mathsf{X}g))}) \\ G = & \mathsf{GF}(s) \rightarrow \mathsf{G}(s \rightarrow \mathsf{X}g). \end{cases}$$

We have 'patched' $\mathcal{C}_2$ by creating a new contract $\mathcal{C}_2'$ that can substitute it. The contract $\mathcal{C}_2'$ can now require the robot to greet on the step after it sees a person, under the assumptions of $\mathcal{C}_2$. The process of generating $\mathcal{C}_2'$ was fully automated. It did not require the designer to look at the specifications and make manual adjustments, which can be hard to do as the complexity of the specifications increases.

IX. Discussion

This section discusses trade-offs that the designer might consider when using CR^2. We consider (i) whether or not the time-steps between the abstract and concrete maps should be the same and (ii) whether the choice of candidate compositions should be based on the highest or lowest refinement scores.

a) Time Step Duration: Let us consider the specification $\mathsf{OrderedPatrolling}(l_b, l_f, l_e)$, which requires the robot to patrol in the abstract map locations L_B, L_F, L_E in the order ($L_B \rightarrow L_F \rightarrow L_E$). Specifically, the robot must move away from L_B, L_F, or L_E immediately after (in the next step) they have been visited and cannot return to the same location before having finished the patrolling of all three. This specification is consistent in the abstract domain. However, it can not be 'refined' by any library of components defined over the concrete map. This is the problem: after visiting L_3, in the next time step the robot should leave L_F without going back to L_5. However since L_F is *covered* by L_1, L_3, and L_4, the robot is stuck and can never reach L_2 (i.e., L_E in the abstract map), thus failing to realize the specification.

CR^2 can help the designer to identify such problems and automatically repair the specification. For example, a candidate composition that prescribes the patrolling of locations $L_5 \rightarrow L_3 \rightarrow L_4 \rightarrow L_2$ can be used to *repair* the abstract specification. This would result in a more *relaxed* Patrolling of locations, i.e., one that does not require a strict order. However, a designer might want to consider that a time step in the abstract map has 'a different duration' from a time step in the concrete map. Instead of letting CR^2 relax the specification by removing the order among locations, the designer could manually repair the specification by, for example, substituting each '*next*' (X) operator with as many next operators as the number of locations in the concrete map. This would ensure that the robot has time to leave the front area in the more concrete description of the store.

b) Refinement Score: When looking for implementations in our library that meet the top-level contract, our framework prefers the most refined implementation possible, i.e., the implementation supporting as many features as possible. One could argue that this would likely be the most expensive implementation and that, thus, one would prefer the least feature-rich implementation. Our framework can be extended to support this implementation, too.

When the library cannot immediately refine the top-level specification, we argue that the choice of which candidate composition to choose (i.e., the one with the highest or the lowest refinement score) comes at a trade-off with the strategy adopted (i.e., search or repair). If the strategy is to search for missing components, one would like to have a candidate with the highest refinement score, as this would be a solution that delegates as little functionality as possible to the missing specification that needs to be implemented with an external library. On the other hand, if the strategy is to repair, one could select the composition with the lowest refinement score. By choosing the composition with the least functionality, the '*patch*' that we are applying to the original contract (after performing contract separation and merging) will be 'lighter' (i.e., less demanding) than a repair performed by a more refined candidate composition.

X. Conclusions

We presented a contract-based framework for modeling and refining robotic mission specifications using libraries of mission components at various abstraction layers. When the refinement of a specification is not possible out of the current library, we provided a method to automatically repair the specification, so that it can be refined using the library, or effectively guide the search for new implementations that can refine it. Our methodology is fully automated and based on contract manipulations via the quotient, separation, and merging operations. We implemented our framework in the tool CR^2. As future work, we plan to test it on a large-scale case study and further investigate the systematic generation of libraries for robotic mission specification.

References

[1] Alur, R., Moarref, S., Topcu, U.: Counter-strategy guided refinement of GR (1) temporal logic specifications. In: 2013 Formal Methods in Computer-Aided Design. pp. 26–33. IEEE (2013)

[2] Autili, M., Grunske, L., Lumpe, M., Pelliccione, P., Tang, A.: Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar. IEEE Transactions on Software Engineering **41**(7), 620–638 (2015)

[3] Baier, C., Katoen, J.P.: Principles of model checking. MIT press (2008)

[4] Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., et al.: Contracts for system design. Foundations and Trends in Electronic Design Automation **12**(2-3), 124–400 (2018)

[5] Brizzio, M., Degiovanni, R., Cordy, M., Papadakis, M., Aguirre, N.: Automated repair of unrealisable ltl specifications guided by model counting. arXiv preprint arXiv:2105.12595 (2021)

[6] Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: CAV. pp. 334–342 (2014)

[7] Cavezza, D.G., Alrajeh, D.: Interpolation-based gr (1) assumptions refinement. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 281–297. Springer (2017)

[8] Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Environment assumptions for synthesis. In: International Conference on Concurrency Theory. pp. 147–161. Springer (2008)

[9] Chatzieleftheriou, G., Bonakdarpour, B., Smolka, S.A., Katsaros, P.: Abstract model repair. In: NASA Formal Methods Symposium. pp. 341–355. Springer (2012)

[10] Chen, J., Sun, R., Kress-Gazit, H.: Distributed control of robotic swarms from reactive high-level specifications. In: 2021 IEEE 17th International Conference on Automation Science and Engineering (CASE). pp. 1247–1254 (2021)

[11] Damm, W., Hungar, H., Josko, B., Peikenkamp, T., Stierand, I.: Using contract-based component specifications for virtual integration testing and architecture design. In: 2011 Design, Automation & Test in Europe. pp. 1–6. IEEE (2011)

[12] Ergurtuna, M., Yalcinkaya, B., Gol, E.A.: An automated system repair framework with signal temporal logic. Acta Informatica pp. 1–27 (2021)

[13] Fainekos, G., Kress-Gazit, H., Pappas, G.: Temporal logic motion planning for mobile robots. In: Proceedings of the 2005 IEEE International Conference on Robotics and Automation. pp. 2020–2025 (2005)

[14] Fainekos, G.E., Girard, A., Kress-Gazit, H., Pappas, G.J.: Temporal logic motion planning for dynamic robots. Automatica **45**(2), 343–352 (2009)

[15] Finkbeiner, B.: Synthesis of reactive systems. In: Esparza, J., Grumberg, O., Sickert, S. (eds.) Dependable Software Systems Engineering. NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 45, pp. 72–98. IOS Press (2016)

[16] Finucane, C., Jing, G., Kress-Gazit, H.: LTLMoP: Experimenting with language, temporal logic and robot control. In: International Conference on Intelligent Robots and Systems (IROS). pp. 1988–1993. IEEE (2010)

[17] Gaaloul, K., Menghi, C., Nejati, S., Briand, L.C., Wolfe, D.: Mining assumptions for software components using machine learning. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 159–171 (2020)

[18] Ghosh, S., Sadigh, D., Nuzzo, P., Raman, V., Donzé, A., Sangiovanni-Vincentelli, A.L., Sastry, S.S., Seshia, S.A.: Diagnosis and repair for synthesis from signal temporal logic specifications. In: Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control. pp. 31–40 (2016)

[19] Guo, M., Dimarogonas, D.V.: Multi-agent plan reconfiguration under local LTL specifications. The International Journal of Robotics Research (2015)

[20] Iannopollo, A., Nuzzo, P., Tripakis, S., Sangiovanni-Vincentelli, A.: Library-based scalable refinement checking for contract-based design. In: 2014 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1–6. IEEE (2014)

[21] Incer, I., Sangiovanni-Vincentelli, A., Lin, C.W., Kang, E.: Quotient for assume-guarantee contracts. 2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2018 (2018)

[22] Kress-Gazit, H., Fainekos, G.E., Pappas, G.J.: Temporal-logic-based reactive mission and motion planning. IEEE Transactions on Robotics **25**(6), 1370–1381 (2009)

[23] Li, W., Dworkin, L., Seshia, S.A.: Mining assumptions for synthesis. In: Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMPCODE2011). pp. 43–50. IEEE (2011)

[24] Mallozzi, P., Nuzzo, P., Pelliccione, P.: Incremental refinement of goal models with contracts. In: in submission to Fundamentals of Software Engineering (FSEN) 2021. IEEE (2020)

[25] Mallozzi, P., Nuzzo, P., Pelliccione, P., Schneider, G.: Crome: Contract-based robotic mission specification. In: 2020 18th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE). IEEE (2020)

[26] Maniatopoulos, S., Schillinger, P., Pong, V., Conner, D.C., Kress-Gazit, H.: Reactive high-level behavior synthesis for an atlas humanoid robot. In: 2016 IEEE International Conference on Robotics and Automation (ICRA). pp. 4192–4199. IEEE (2016)

[27] Maoz, S., Ringert, J.O.: GR(1) synthesis for LTL specification patterns. In: Foundations of Software Engineering (FSE). ACM (2015)

[28] Maoz, S., Ringert, J.O.: Synthesizing a Lego Forklift Controller in GR(1): A Case Study. In: Proceedings Fourth Workshop on Synthesis (SYNT) (2015)

[29] Maoz, S., Ringert, J.O.: On well-separation of GR(1) specifications. In: Foundations of Software Engineering (FSE). ACM (2016)

[30] Maoz, S., Ringert, J.O., Shalom, R.: Symbolic repairs for gr (1) specifications. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). pp. 1016–1026. IEEE (2019)

[31] Menghi, C., Tsigkanos, C., Pelliccione, P., Ghezzi, C., Berger, T.: Specification Patterns for Robotic Missions. IEEE Transactions on Software Engineering pp. 1–1 (2019)

[32] Menghi, C., Garcia, S., Pelliccione, P., Tumova, J.: Multi-robot LTL Planning Under Uncertainty. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E. (eds.) Formal Methods. pp. 399–417. Springer International Publishing, Cham (2018)

[33] Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: Explicit reactive synthesis strikes back! In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10981, pp. 578–586. Springer (2018)

[34] Nuzzo, P., Finn, J., Iannopollo, A., Sangiovanni-Vincentelli, A.L.: Contract-based design of control protocols for safety-critical cyber-physical systems. In: Proc. Design Automation and Test in Europe Conference. pp. 1–4 (Mar 2014)

[35] Nuzzo, P., Li, J., Sangiovanni-Vincentelli, A.L., Xi, Y., Li, D.: Stochastic assume-guarantee contracts for cyber-physical system design. ACM Trans. Embed. Comput. Syst. **18**(1), 2:1–2:26 (Jan 2019). https://doi.org/10.1145/3243216, http://doi.acm.org/10.1145/3243216

[36] Nuzzo, P., Lora, M., Feldman, Y.A., Sangiovanni-Vincentelli, A.L.: CHASE: Contract-based requirement engineering for cyber-physical system design. In: 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 839–844. IEEE (2018)

[37] Nuzzo, P., Sangiovanni-Vincentelli, A., Bresolin, D., Geretti, L., Villa, T.: A platform-based design methodology with contracts and related tools for the design of cyber-physical systems. Proc. IEEE **103**(11) (Nov 2015)

[38] Nuzzo, P., Xu, H., Ozay, N., Finn, J.B., Sangiovanni-Vincentelli, A.L., Murray, R.M., Donzé, A., Seshia, S.A.: A contract-based methodology for aircraft electric power system design. IEEE Access **2**, 1–25 (2014)

[39] Oh, C., Lora, M., Nuzzo, P.: Quantitative verification and design space exploration under uncertainty with parametric stochastic contracts. In: Proceedings of the 41th International Conference on Computer-Aided Design. ICCAD '22 (2022)

[40] Pacheck, A., Konidaris, G., Kress-Gazit, H.: Automatic encoding and repair of reactive high-level tasks with learned abstract representations. In: Accepted, Robotics Research: the 18th Annual Symposium (2019)

[41] Pacheck, A., Moarref, S., Kress-Gazit, H.: Finding missing skills for high-level behaviors. In: 2020 IEEE International Conference on Robotics and Automation (ICRA). pp. 10335–10341. IEEE (2020)

[42] Passerone, R., Incer, I., Sangiovanni-Vincentelli, A.L.: Coherent extension, composition, and merging operators in contract models for system design. ACM Trans. Embed. Comput. Syst. **18**(5s) (Oct 2019)

[43] Sangiovanni-Vincentelli, A., Damm, W., Passerone, R.: Taming Dr. Frankenstein: Contract-based design for cyber-physical systems. European journal of control **18**(3), 217–238 (2012)

[44] Shoukry, Y., Nuzzo, P., Balkan, A., Saha, I., Sangiovanni-Vincentelli, A.L., Seshia, S.A., Pappas, G.J., Tabuada, P.: Linear temporal logic motion planning for teams of underactuated robots using satisfiability modulo convex programming. In: Proc. Int. Conf. Decision and Control (Dec 2017)

[45] Wang, T.E., Daw, Z., Nuzzo, P., Pinto, A.: Hierarchical Contract-Based Synthesis for Assurance Cases. In: NASA Formal Methods Symposium. pp. 175–192. Springer (2022)