CROME: Contract-Based Robotic Mission Specification

Piergiuseppe Mallozzi[1], Pierluigi Nuzzo[3], Patrizio Pelliccione[1,2], Gerardo Schneider[1]
[1]Chalmers University of Technology | University of Gothenburg, Sweden
[2]University of L'Aquila, Italy
[3]Viterbi School of Engineering, University of Southern California, Los Angeles, USA
Email: mallozzi@chalmers.se, nuzzo@usc.edu, patrizio.pelliccione@univaq.it, gersch@chalmers.se

***Abstract*—We address the problem of automatically constructing a formal robotic mission specification in a logic language with precise semantics starting from an informal description of the mission requirements. We present CROME (Contract-based RObotic Mission spEcification), a framework that allows capturing mission requirements in terms of goals by using specification patterns, and automatically building linear temporal logic mission specifications conforming with the requirements. CROME leverages a new formal model, termed Contract-based Goal Graph (CGG), which enables organizing the requirements in a modular way with a rigorous compositional semantics. By relying on the CGG, it is then possible to automatically: i) check the *feasibility* of the overall mission, ii) further *refine* it from a library of pre-defined goals, and iii) *synthesize* multiple controllers that implement different parts of the mission at different abstraction levels, when the specification is realizable. If the overall mission is not realizable, CROME identifies mission scenarios, i.e., sub-missions that can be realizable. We illustrate the effectiveness of our methodology and supporting tool on a case study.**

I. Introduction

In the near future, service robots will increasingly be used to support tasks in everyday life [1], [2], [3], even though existing solutions are often not readily usable [4]. Service robots are "a type of robot that performs useful tasks for humans or equipment excluding industrial automation applications" [5]. The service robotics market is estimated to reach a value of $24 billion by 2022 [3]. However, the robots that we find in the market today are highly specialized to accomplish a specific function. Their use often reduces to clicking a specific button that will trigger the execution of the specific mission the robot is programmed for. For instance, this is the case of commercial vacuum cleaner robots like Roomba [6]. On the other hand, the advent of multipurpose service robots, required to accomplish various domain-specific missions, calls for new languages and tools to enable end users to accurately specify complex missions [2], [7], [8].

We make a distinction between a *mission requirement*, i.e., an informal description of the mission the robots must perform, and a *mission specification*, i.e., a formulation of the mission in a formal (logical) language with precise semantics [2]. Producing a mission specification from a mission requirement is identified as the *mission specification problem*. 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) [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22]. Using formal languages makes behavioral specifications precise and unambiguous. However, logic formulas can be difficult to interpret for the end user, and generating them can be an error-prone process [23], [24], [2]. Mission requirements, on the other hand, are often ambiguous [25], [26], [27] and make it hard to assess the correctness of the specification [28], [29], [30], [31]. In recent years, there have been many proposals for describing mission requirements based on: *i)* domain specific languages [7], [32], [33], *ii)* natural language [26], and *iii)* visual and end-user-oriented environments [34], [35], [36], [37], mostly used for educational purposes. While the approaches above provide substantial contributions to the mission specification problem, solutions that can scale to complex missions and enable the deployment of service robots in everyday life are still elusive.

As stated in the Multi-Annual Roadmap for Robotics in Europe (MAR) [38], to reduce costs and establish a vibrant component market, we need tools that can support mission *reuse* and diversification, as well as the *variability* of conditions and application scenarios occurring in a real mission. This is also witnessed by our findings during a collaboration with practitioners in the robotic domain [39]. While it is often not difficult to define what the robots should do, the challenge is in coping with the variability of the environments in which the robots operate, especially those involving humans [39]. To address this issue, we would need to explicitly enumerate all the possible variants of a mission using, e.g., state diagrams or flow charts, which can be difficult, tedious, and error-prone.

Menghi et al. [2] have recently identified and proposed a catalogue of robotic movement patterns, which are solutions to recurring problems in mission specification. Patterns are based on LTL formulas, which are often used to automatically synthesize plans [12], [15], [40], [41], [42]. While grounded in a formal language, these patterns can also be used by non-experts and there exist tool support to compose them via conjunction or disjunction for the specification of complex missions [43]. Garcia et al. [7] introduce more complex composition rules, including control-flow operators like fall-back (to define alternative strategies when the previous ones fail), exception-handling (to stop the current execution when

978-1-7281-9148-5/20/$31.00 © 2020 IEEE

an exception is raised and then continue with the current task), and sequence (to perform a sequence of tasks). However, control-flow operators are implemented in software but lack a formal representation in logic, which makes it difficult to verify the feasibility of the entire mission specification in a way that is independent of the implementation.

In this paper, we propose a framework, named CROME (Contract-based RObotic Mission spEcification), that explicitly addresses the problems of *specification reuse* and *environment modeling* in mission specification, enabling the designer to cope with the variability of the application scenarios of a robotic mission. By building on recent work on contract-based requirement engineering [40], [44], [41], leveraging *context-aware contract models* and patterns to generate controller specifications, we decouple the task specification from the specification of the context in which the task is executed. End users explicitly specify the various mission tasks together with their contexts. The overall mission is then automatically compiled by CROME. CROME contributes to the following aspects of the mission specification process:

- *Formulating mission requirements.* We model each requirement as a *goal*, expressed using a set of previously proposed patterns [2], [45]. Goal models have been used over the years as an intuitive and effective means to capture the designer's objectives and their hierarchical structure [46]. In this paper, we augment the notion of goal to explicitly include a concept of *context*, which enables building mission specifications that are adaptable to different environmental conditions. Contexts help capture the variability associated with a mission goal, so that the same goal can be implemented in different ways when used in different contexts.
- *Generating mission specifications.* We introduce a novel model, termed *contract-based goal graph* (CGG), which is automatically generated to formalize a mission and its sub-missions. The CGG is a graph of goals where the root node represents the overall mission, its immediate children represent mission *scenarios*, and the rest of the nodes are part of the sub-missions. In a CGG, goals are captured by assume-guarantee contracts [47] and are linked together using operations and relations between contracts. We differentiate the scenario nodes from other nodes since they are goals that have mutually exclusive contexts and identify sub-missions that cannot be jointly realized.
- *Refining mission specifications out of a library of goals*. We introduce an algorithm that automatically refines the leaf nodes of a CGG using the goals in a library, so that "abstract" goals in the CGG can be further implemented (refined) by more "concrete" goals.

By formalizing the mission specification with a CGG, CROME also offers the following capabilities:

- *Requirement conflict identification*. By checking the satisfiability of the CGG contracts, we are able to identify the presence of conflicts in the mission requirements and immediately inform the designer, before attempting at synthesizing a controller.
- *Realizability checking and controller generation*. CROME checks the realizability of each scenario in the CGG and informs the designer of which sub-goals can be realized (i.e., a controller can be synthesized), given a model of the environment. For each realizable goal of the CGG, CROME synthesizes a controller in the form of a Mealy machine. The controllers are produced together with the CGG.

Our case study shows that the modularity of the CGG allows efficiently checking the feasibility of a mission. The identification of the scenarios allows analyzing the impact of environment variability on the realizability of the robotic mission. The automatic refinement from a goal library facilitates the reuse of existing goals to implement complex specifications. Finally, mutually exclusive scenarios can point to control architectures that may not have a centralized implementation, while still being realizable in a decentralized fashion.

The rest of the paper is organized as follows. In Section II we provide background notions and related work on contracts, linear temporal logic, specification patterns, and contexts. We introduce CROME in Section III. We detail how the robotic mission is specified and mutually exclusive contexts are generated in Section IV and Section V, respectively. We present the CGG in Section VI and illustrate our approach on a case study motivated by a care center in Section VII. Finally, in Section VIII, we draw some conclusions.

II. Background and Related Work

We provide some background on the basic building blocks of CROME: contracts, linear temporal logic, and specification patterns.

A. Assume-Guarantee Contracts

Contract-based design [47], [44] has emerged over the years as a design paradigm capable of providing formal support for building complex systems in a modular way, by enabling compositional reasoning, stepwise refinement, and reuse of pre-designed components.

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 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 that are 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 logic formula; we then use the formula itself to denote them, with a slight abuse of notation, whenever there is no confusion. 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 forms 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. 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.

1) Contract 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. $\mathcal{C}$ 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.

2) Contract Composition: Contracts associated with distinct implementations can be combined via the composition operation ($\|$) to specify the composition between the corresponding implementations. 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:

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

$$G = G_1 \cap G_2. \tag{2}$$

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 $\mathcal{C}$ [47].

3) Contract Conjunction: Different contracts on a single implementation can be combined using the conjunction operation ($\wedge$). Let $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$ be two contracts. We can compute their conjunction by taking the greatest lower bound of $\mathcal{C}_1$ and $\mathcal{C}_2$ with respect to the refinement relation. Intuitively, the conjunction $\mathcal{C} = \mathcal{C}_1 \wedge \mathcal{C}_2$ is the weakest (most general) contract that refines both $\mathcal{C}_1$ and $\mathcal{C}_2$. $\mathcal{C}$ can be computed by taking the intersection of the guarantees and the union of the assumptions, that is:

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

Intuitively, an implementation satisfying $\mathcal{C}$ must satisfy the guarantees of both $\mathcal{C}_1$ and $\mathcal{C}_2$, while being able to operate in either of the environments of $\mathcal{C}_1$ or $\mathcal{C}_2$.

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:

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

where φ, φ_1, and φ_2 are LTL formulas, $\overline{\varphi}$ is the negation of φ, $\vee$ is the logic disjunction, $\mathbf{X}$ is the temporal operator *next* and $\mathbf{U}$ is the temporal operator *until*. Other temporal operators such as *globally* ($\mathbf{G}$) and *eventually* ($\mathbf{F}$) can be derived as follows: $\mathbf{F}\ \varphi = \mathtt{true}\ \mathbf{U}\ \varphi$ and $\mathbf{G}\ \varphi = \overline{\mathbf{F}\ \overline{\varphi}}$. We refer to the literature [48] for the formal semantics of LTL.

C. Specification Patterns and Context

1) Robotic Patterns: Robotic patterns have been proposed as a solution to recurrent mission specification problems based on the analysis of mission requirements in the robotic literature [2]. CROME supports 22 patterns [2], capturing robot movements and actions performed as a robot move in the environment, organized into three groups: *core movement* patterns, *triggers*, and *avoidance* patterns.

For example, let us assume that the mission requirement is: ‘A robot must patrol a set of locations in a certain strict order.’ The designer can formulate this requirement by using the *Strict Ordered Patroling* pattern, instantiated for the required set of locations. Let l_1, l_2, and l_3 be the atomic propositions of type *location* that the robot must visit in the given order. The mission requirement can then be reformulated as ‘Given the locations l_1, l_2, and l_3, the robot should visit all the locations indefinitely and following a strict order,’[1] leading to the following LTL formulation:

$$\begin{aligned} &\mathbf{G}(\mathbf{F}(l_1 \wedge \mathbf{F}(l_2 \wedge \mathbf{F}(l_3)))) \wedge (\overline{l_2}\ \mathbf{U}\ l_1) \wedge (\overline{l_3}\ \mathbf{U}\ l_2) \\ &\wedge \mathbf{G}(l_2 \rightarrow \mathbf{X}(\overline{l_2}\ \mathbf{U}\ l_1)) \wedge \mathbf{G}(l_3 \rightarrow \mathbf{X}(\overline{l_3}\ \mathbf{U}\ l_2)) \\ &\wedge \mathbf{G}(l_1 \rightarrow \mathbf{X}(\overline{l_1}\ \mathbf{U}\ l_3)) \\ &\wedge \mathbf{G}(l_1 \rightarrow \mathbf{X}(\overline{l_1}\ \mathbf{U}\ l_2)) \wedge \mathbf{G}(l_2 \rightarrow \mathbf{X}(\overline{l_2}\ \mathbf{U}\ l_3)). \end{aligned} \tag{3}$$

As shown in this example, a robotic pattern can significantly facilitate the difficult and error-prone task of mission specification.

2) Specification Patterns with Scopes: Our library of patterns is also inspired by the work of Dwyers et al. [45], who developed a catalogue of generic property specification patterns for a broader range of applications. In particular, we adopt the notion of *scope*, which provides a way to define the extent to which a property must hold [45]. For example, for

[1] http://roboticpatterns.com/pattern/strictorderedpatrolling/

the *universality* pattern, in which we require that a property e be true, we can introduce the following scopes:

$$e \text{ global} = \mathbf{G}(e) \quad (4)$$
$$e \text{ before } r = \mathbf{F}(r) \rightarrow (e \ \mathbf{U} \ r) \quad (5)$$
$$e \text{ after } q = \mathbf{G}(q \rightarrow \mathbf{G}(e)) \quad (6)$$
$$e \text{ between } q \text{ and } r = \mathbf{G}((q \ \wedge \ \overline{r} \ \wedge \ \mathbf{F} \ r) \rightarrow (e \ \mathbf{U} \ r)) \quad (7)$$
$$e \text{ after } q \text{ until } r = (\mathbf{G}(q \ \wedge \ \overline{r} \rightarrow ((e \ \mathbf{U} \ r) \mid \mathbf{G} \ p))), \quad (8)$$

where q, r are also properties or events. The patterns proposed by Dwyers et al. [45] were also extended to incorporate time [49] and probability [50]. Autili et al. [51] present a unified catalogue of property specification patterns including, among others, the patterns mentioned above [45], [49], [50]. A description of the patterns in this catalogue [51] is also available online [52].

3) Context: Many characterizations of the 'context' of an application have been provided, often informally, in the literature. In context-aware ubiquitous computing [53], the context of an application may include information like location, identities of nearby people and objects, time of the day, season, or temperature. More generally, Dey and Abowd [54] define context as "any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves." In the robotic domain, Bloisi et al. [55] define mission-related contexts as "choices that are useful in robotic scenarios for adapting the robot behaviors to the different situations." CROME builds on these characterizations and adapts them to robotic missions by formalizing a context as a property associated with a goal and expressed by a logic formula.

III. Overview of CROME

Figure 1 shows the mission specification process with CROME, which can be summarized as follows.

Phase (1): A robotic mission can be decomposed into a set of requirements prescribed to a system (the robot) acting in an environment. In the requirement capture phase, the designer provides the inputs to CROME: specification goals, domain properties, and a goal library. Each specification *goal* $\mathcal{G}_i$, modeling a mission requirement, is specified using a *pattern* p_i and instantiated in a *context* x_i. The domain properties encapsulate constraints on the environment and the system. They belong to three categories: (1) physical rules, denoted by t, e.g., specifying the map of a building; (2) logical rules, denoted by l, e.g., specifying logical partitions (areas) of a building; and (3) system constraints, denoted by s, e.g., specifying actions that can not be performed simultaneously by the robotic system, such as going in two locations simultaneously. Finally, the designer can provide a library of predefined goals that will be used in phase (4) to automatically refine the mission specification.

Phase (2): In this phase, CROME analyzes the relationship between goals and groups them in clusters based on their context. For example, two goals are mutually exclusive if the conjunction of their contexts produces a contradiction with respect to the domain properties. CROME produces clusters of goals and associates to each cluster a new generated context x'_j, which is guaranteed to be mutually exclusive with any other cluster's context. This phase is detailed in Section V.

Phase (3): CROME builds the Contract-based Goal Graph (CGG), a formal model representing a graph of goals. CROME formulates *assume-guarantee (A/G) contracts* for each goal and leverages the clusters created in phase (2) to determine the structure of the CGG, which in this phase takes the form of a tree. Contracts belonging to the same cluster are combined using composition; the resulting contracts of the clusters are combined using conjunction. By building the CGG, CROME analyzes the consistency of the mission and helps detect conflicting requirements.

Phase (4): Given a library of goals $\mathcal{L}$, for each leaf node of the CGG, CROME checks whether it can be refined by a goal of the library. Checking refinement between goals amounts to checking refinement between the contracts formalizing the goals. If a library goal is found, it is connected to the CGG and becomes a new leaf node to be scheduled for refinement. The refinement procedure continues recursively until no more library goals can be used to extend the CGG. A library goal can refine multiple goals of the CGG. In this case, we introduce a new leaf for the library goal and connect it with all the goals it refines, which makes the CGG no longer a tree.

Phase (5): CROME checks the realizability of each node of the CGG. If the root of the CGG $\mathcal{M}$ is realizable, then a controller can be generated for the whole mission specified by the designer. If the root node is not realizable, then it may still be possible to realize some of the mission scenarios, as identified in the CGG. Moreover, for each scenario, different controllers can be generated at different abstraction levels. We provide details for this phase in Section VI-B3.

IV. Capturing Mission Requirements

Mission requirements are provided by the designer in terms of goals and domain properties, expressed in a structured way using patterns and scopes, which will be automatically translated into LTL formulas. Goals and domain properties are defined over a set of atomic propositions AP, which can be true or false at any point during the mission. In the following, we detail the building blocks of the mission requirements, i.e., atomic propositions, contexts, goals, and domain properties.

A. Atomic Propositions

Atomic propositions (APs) can be grouped into six categories based on the semantics associated with them. *Sensor* APs, *location* APs, and *action* APs are associated with the robot. *Location-context* APs, *time-context* APs, and *identity-context* APs are associated with the context. APs can refer to *controlled* or *uncontrolled* variables for the robot. For example, location APs and action APs refer to controlled variables, since a robot can choose its next location and action, while the other

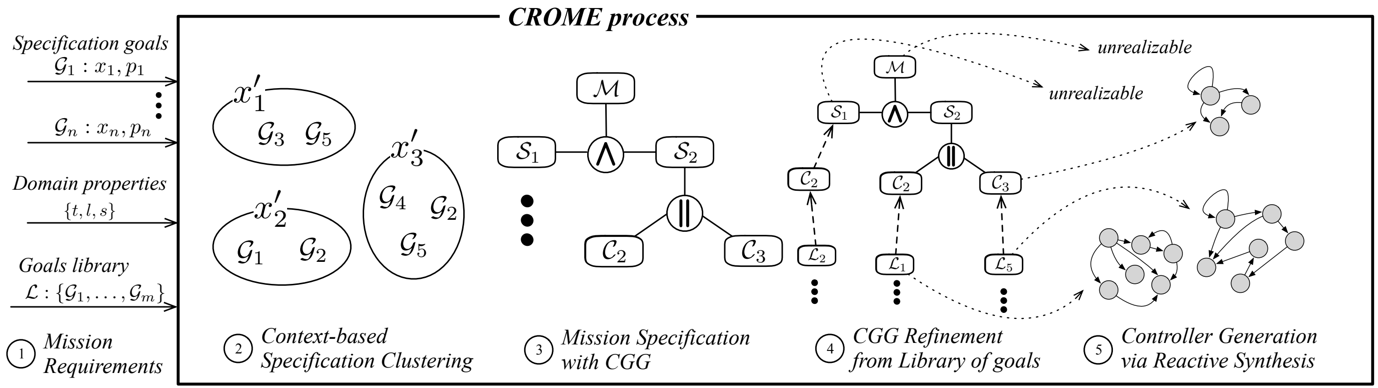

Fig. 1: Mission specification process

APs refer to uncontrolled variables, since they relate to the context or the perception of the environment.

B. Context

We formalize *contexts* in terms of Boolean predicates encoding the situation in which a goal must be active. For example, context propositions can encode information related to the location, time, or identities associated with a goal. In a robotic application, locations specify *where* a robot can be, time specifies *when* a certain goal must be active (e.g., during the day or the night), and identities specify the state of external entities (*who*) that may interact with the robot.

C. Goals

In CROME each mission requirement is modeled by a goal, characterized by the following elements:

- *Name*: goal identifier;
- *Description*: English description of the mission requirement;
- *Context*: Boolean predicates over the context APs that hold true for the goal;
- *Objective*: formulas over all the AP expressing what the robot must achieve under the context of the goal.

Goal objectives can be expressed, for example, by properties including atomic propositions in combination with Boolean operators and the temporal operator $\mathbf{G}$ (globally), which suits a large number of natural-language requirements [56]. However, CROME enables the expression of more complex objectives. We address the generation of complex temporal logic formulas via the robotic patterns [2] and the specification patterns with scopes [45] in Section II-C.

D. Domain Properties

Domain properties are general constraints that must hold for the whole mission; they can relate to the robotic agent or the environment and use any type of AP. Domain properties can also be generated by using patterns or basic logic predicates over the APs. CROME accepts three kinds of domain properties, which are all compiled as logic predicates:

- *Mutex properties* relate predicates that cannot be true at the same time. For example, `warehouse` and `shop` AP can be marked as mutex propositions, since they represent separate physical environments where the robot cannot be at the same time.
- *Inclusion properties* express constraints on pairs of propositions, predicates, or patterns, such that, when the first term is true, then also the second term must be true. For example, *SequencedPatroling*(`cashier`, `entrance`, `warehouse`) and *Patroling*(`shop`) can be part of an inclusion property since whenever the robot patrols the `cashier`, `entrance`, and `warehouse` locations in sequence, then it patrols the `shop`.
- *Adjacency properties* express constraints over location APs that are adjacent, i.e., such that one location can be reached within one step from another location. For example, adjacency properties can be used to describe the grid-map of the environment, eventually constraining the movements of the robotic system.

Separating the domain properties from the goals is instrumental to mission specification reuse, as it allows instantiating the same goals in different environments enjoying different domain properties.

V. Context-Based Specification Clustering

In this phase, CROME groups the goals into separate *clusters* based on their contexts. If the contexts associated with two goals are jointly satisfiable, then the goals are placed in the same cluster; otherwise they are placed in different clusters, which are marked as mutually exclusive. A cluster is then a tuple containing a mutex-context and a set of goals.

Algorithm 1 automates this phase. It takes as inputs the list of goals and the domain properties, referred as `rules`. The result is a set of clusters, each associated with a new *mutex-context*, which is inconsistent with any mutex-context associated with another cluster. First, the algorithm extracts all the contexts from the list of goals and computes all the possible combinations of contexts. For each combination of contexts `comb` the algorithm performs the following operations:

- *Saturation*: it adds to the combination the negation of all the contexts that are not part of the combination.

Algorithm 1: Extract mutually exclusive context clusters

```
Input: goals: list of goals, rules: domain properties
Output: clusters: set of tuples, where each tuple contains a
        mutex-context and a set of goals
goals_cxts ← extract_context(goals)
mtx_cxts ← ∅
/* Compute all the possible combinations for L
   goals_cxts                                              */
for i in {0..L} do
    /* Extract all combinations of i contexts
       */
    comb_i ← combinations(contexts, i)
    /* For each combination                                */
    for comb in comb_i do
        for ctx in contexts do
            /* Saturate the combination                    */
            /* If the context is not part of the
               combination                                 */
            if ctx not in comb then
                /* Get the negation of the context
                   formula                                 */
                ctx_neg ← Not(ctx)
                /* Add the negation to the
                   combination                             */
                comb_i ← comb_i ∪ ctx_neg
        /* Add additional rules when needed                */
        for r in rules do
            if r applies to comb then
                comb ← comb ∪ r
        /* Simplify formulas in comb (1)                   */
        comb ← simplify(comb)
        if comb is consistent then
            /* Conjoin all the elements in comb
               and save the result in
               new_contexts                                */
            mtx_cxts ← mtx_cxts ∪ And(comb)
/* Group contexts in mtx_cxts (2)                          */
mtx_cxts ← group(mtx_cxts)
clusters ← ∅
for cxt in mtx_cxts do
    /* Map goals to contexts in mtx_cxts (3)               */
    c_to_g = map(ctx, goals)
    clusters ← clusters ∪ c_to_g
/* Select final clusters (4)                               */
clusters ← select(clusters)
return clusters
```

- *Adding rules*: if any predicate in comb contains APs that are also in a predicate of the rules, then it adds the predicate in the rules to the combination.
- *Consistency check*: if the conjunction of all the contexts in a combination, after the addition of the rules predicate is consistent, meaning that the resulting formula is satisfiable, then it produces a new mutex-context, obtained from the conjunction of the contexts augmented with the rules.

Algorithm 1 then groups the mutex-contexts, maps each goal to a mutex-context to form different clusters, and selects the final mutex-contexts among those that are mapped to the same set of goals. Given L contexts, there are at most M

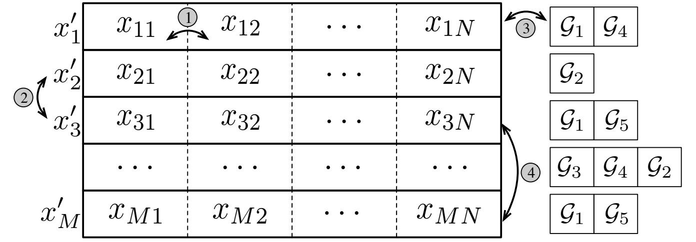

Fig. 2: Example of mapping of 5 goals $\mathcal{G}_1, \ldots, \mathcal{G}_5$ to M mutex-contexts $x'_1, \ldots, x'_M$. Each combination of contexts contains at most N logic propositions in conjunction. A circled number indicates refinement checking tasks among formulas expressing the goal contexts or mutex-contexts.

combinations, where

$$M = \sum_{k=1,\ldots,L} \binom{L}{k} = \sum_{k=1,\ldots,L} \frac{L!}{k!(L-k)!}.$$

Each combination contains at most $N = L + R$ elements in conjunction, where L is the number of contexts and R is the number of additional rules, i.e., domain properties related to the context. For example, Figure 2 shows a list of context combinations that are mapped to a list of goals. Each combination is formed by propositions x_{ij} representing contexts and rules, and their conjunction results in a mutex-context x'_i. Every mutex-context x'_i is then associated with a set of goals $\mathcal{G}$ to form a cluster. We detail below some of the functions used in Algorithm 1.

In CROME, contexts and mutex-contexts are formulas. Therefore, to manipulate contexts, we define a refinement relation between formulas. A formula ϕ refines a formula ψ if and only if $\phi \rightarrow \psi$ is valid. If ϕ refines ψ, then the behaviors satisfying ϕ are included in the set of behaviors satisfying ψ. The *simplify*, *group*, *map*, and *select* functions in Algorithm 1 perform refinement checks among pairs of LTL formulas, as also marked by the circled numbers in Figure 2:

- (1) ***simplify***. Each mutex-context is built as a conjunction of clauses. Each clause x_i represents a context, a negation of a context, or a context rule. For each pair of clauses x_{ia}, x_{ib} in a mutex-context x'_i, Algorithm 1 checks whether x_{ia} refines x_{ib} and removes the most abstract clause, e.g., if $x_{ia} \rightarrow x_{ib}$, then $x_{ia} \wedge x_{ib} = x_{ia}$.
- (2) ***group***. This process is similar to the one in *simplify*, but operates on pairs of mutex-contexts x'_i, x'_j, as shown in Figure 2, rather than the clauses of each mutex-context. For each pair of mutex-context formulas, if a formula implies another one, the group function only retains the most refined one.
- (3) ***map***. The mapping process connects each specification goal to a new mutex-context x'_i, and finally forms a cluster. Let x_i be the context of the specification goal $\mathcal{G}_i$. Then, the map function checks whether all the behaviors satisfying the mutex-context x'_i are contained in context x_i, that is, whether x'_i is a refinement of x_i. If this is the case, CROME links $\mathcal{G}_i$ to the new context x'_i. Because

mutex-contexts are constructed by refining contexts, there must exist a mutex-context x'_i that refines x_i.

- ④ ***select***. It may happen that more than one mutex-context are linked to the same goal. For example, in Figure 2, both x'_3 and x'_M are linked to $\mathcal{G}_1$ and $\mathcal{G}_5$. In this case, CROME maps the goal to the cluster with the most abstract context.

VI. Mission Specification via Contract-Based Goal Graphs

In this step, domain properties and goals are formalized using A/G contracts and organized using a CGG.

A. Contract Formalization and Analysis

Once the clusters and mutex-contexts are identified by Algorithm 1, CROME produces one contract $\mathcal{C}_i$ for each goal $\mathcal{G}_i$ in the clusters, where:

- the assumptions capture the domain properties related to the environment in which the mission is deployed;
- the guarantees capture the properties associated with the goal objectives and the corresponding context via formulas of the form

$$\mathbf{G}(ctx \rightarrow obj) \tag{9}$$

for a context ctx and an objective obj expressed, for example, as a conjunction of robotic patterns.

Since contract assumptions and guarantees are expressed by logic formulas, CROME checks for incompatibility and inconsistency (i.e., emptiness of assumptions or guarantees) by checking whether the logic formulas are satisfiable. Moreover, CROME performs a *feasibility check* to verify whether contracts are well-formed, i.e., whether $A \cap G \neq \emptyset$ holds. For example, the contract $\mathcal{C} = (\phi_a, \phi_g)$, where $\phi_a := \mathbf{G}(\texttt{env})$, $\phi_g := \mathbf{G}(\texttt{env}) \rightarrow (\mathbf{F}\texttt{move} \;\; \wedge \;\; \overline{\mathbf{F}\texttt{move}})$, and $\texttt{env}$ and $\texttt{move}$ are APs, is compatible and consistent. However, $\mathcal{C}$ is not well-formed since $\phi_a \wedge \phi_g$ is infeasible. LTL satisfiability checks can be reduced to model checking problems [44], [57]. We check the satisfiability of a formula ϕ by querying a model checker for the validity of $\psi := \neg\phi$. If ψ is valid, then ϕ is unsatisfiable. A counterexample invalidating ψ is a model, i.e., a satisfying trace, for ϕ.

B. Contract-Based Goal Graph

A CGG, shown in Figure 3, is a graph $T = (\Upsilon, \Sigma)$, where each node $\upsilon \in \Upsilon = \Gamma \cup \Delta$ is either a *goal node* $\gamma \in \Gamma$ or an *operator node* $\delta \in \Delta$, with $\Gamma \cap \Delta = \emptyset$. Each goal node is the formalization of a goal via a contract. Each operator node takes a value in $\{\|, \wedge\}$ and represents an operation (composition or conjunction) between contracts. Each edge $\sigma \in \Sigma$ can connect a goal node in Γ to an operator node in Δ or two goal nodes. In the former case, the edge is a *connection link*. Otherwise, it is a *refinement link*.

Any goal node of the CGG can be realized to achieve a controller. A realization of the root node covers the whole mission but the synthesis problem could be infeasible or intractable in practice. The decomposition of the goals via the modularity of the CGG allows pointing out portions of the mission that may be independently realizable.

1) Building the CGG via Composition and Conjunction: CROME uses contract conjunction and composition to combine the different goals and form the CGG. Composition and conjunction produce more complex goals from simpler ones, which are then connected to the CGG. However, composition demands that the resulting goal operate in environments satisfying the assumptions of all the composing goals. On the other hand, conjunction requires that the resulting goal operate with environments that satisfy either (but not necessarily both) of the assumptions of the original goals. CROME uses contract conjunction to blend different *scenarios* that must be both satisfied by the mission, while the scenarios are built by composition of smaller contracts. More specifically, the CGG is built by computing, for each cluster, the composition of the goals associated with the cluster (e.g., $\mathcal{C}_2$ and $\mathcal{C}_3$ in Figure 3). Goals can be interconnected to form complex structures, where the guarantees of one (or more) goal are used to discharge the assumptions of other goals. Such a composition produces a new goal node (e.g., $\mathcal{S}_1$ in Figure 3), which is the scenario supported by the goals in the cluster. Finally, the overall mission specification $\mathcal{M}$ is compiled in terms of the conjunction of all the mutually exclusive scenarios (e.g., $\mathcal{S}_1$ and $\mathcal{S}_2$ in Figure 3).

2) Extending the CGG via Refinement from Library of Goals: A library of goals is a collection of goals, each formalized with a contract and labeled with a *cost*. CROME can automatically extend a CGG by refining its leaf nodes with goals chosen from a library of goals, while minimizing the overall cost. Figure 3 shows how contracts $\mathcal{C}_1, \mathcal{C}_2$, and $\mathcal{C}_3$ are refined by the library goals $\mathcal{L}_1, \mathcal{L}_2$, and $\mathcal{L}_5$. A refinement link between a library goal $\mathcal{L} = (a_l, g_l)$ and a leaf node $\mathcal{C} = (a_c, g_c)$ is created if and only if $\mathcal{L} \preceq \mathcal{C}$. $\mathcal{L}_2$ is further refined by library goal $\mathcal{L}_1$, while $\mathcal{L}_5$ refines two goal nodes of the CGG, which are both corresponding to contract $\mathcal{C}_3$.

3) Controller Synthesis: CROME checks the realizability of each goal node of the CGG and, if it is realizable, it produces a controller using reactive synthesis. In Figure 3, goal $\mathcal{C}_3$ is refined by $\mathcal{L}_5$ and they can both be realized with two controllers at different abstraction levels. For example, let us assume that $\mathcal{C}_3$ requires as objective *Patroling*(a, b, c), corresponding to the LTL formula $\phi_a = \mathbf{GF}\ a \wedge \mathbf{GF}\ b \wedge \mathbf{GF}\ c$. On the other hand, the library goal objective is *SequencedPatroling*(a, x, y, z, c) corresponding to $\phi_r = \mathbf{GF}(a \wedge \mathbf{F}(x \wedge \mathbf{F}(y \wedge \mathbf{F}(z \wedge \mathbf{F}\ c))))$. Further, there exists a domain property of type inclusion between *SequencedPatroling*(x, y, z) and *Patroling*(b), that is, $\psi = \mathbf{G}(\mathbf{F}(x \wedge \mathbf{F}(y \wedge \mathbf{F}\ z))) \rightarrow \mathbf{GF}\ b$. Given $\mathcal{C}_3 = (\psi, \phi_a)$ and $\mathcal{L}_5 = (\textit{true}, \phi_r)$, we have and $\mathcal{L}_5 \preceq \mathcal{C}_3$, since $\psi \rightarrow \textit{true}$ and $\phi_r \rightarrow (\psi \rightarrow \phi_a)$ are valid formulas.

VII. Case Study: Urgent Care

We consider a mission performed by a service robot working in an urgent care clinic, and consisting of several tasks. Figure 4 shows the map of the clinic together with the contexts.

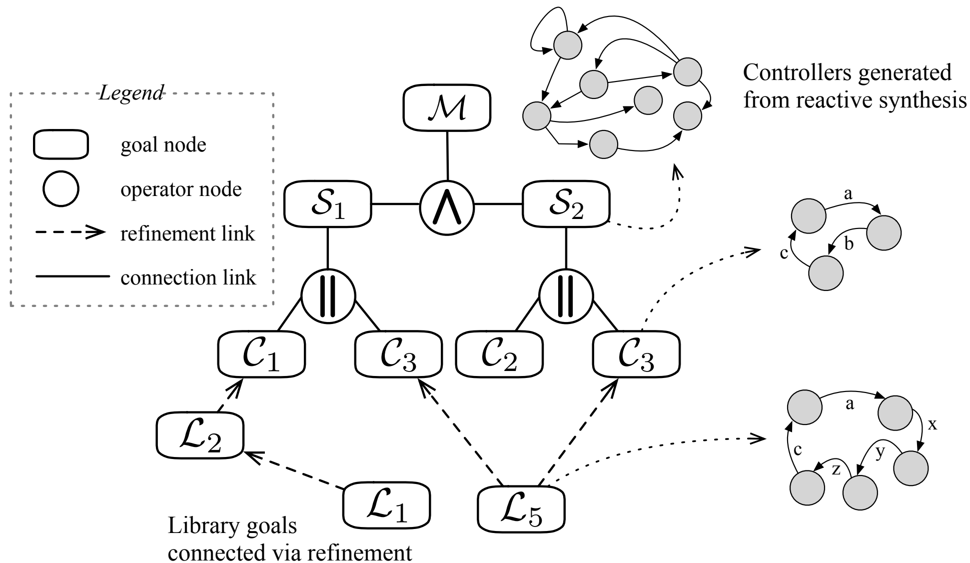

Fig. 3: Example of CGG where some of the goal nodes are linked to a Mealy machine representing the controller synthesized from the node.

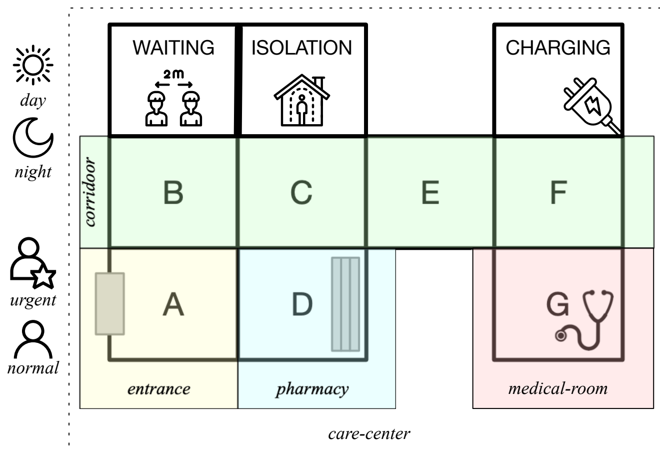

Fig. 4: Care clinic map showing the time, location, and identity contexts (written in italics).

Time contexts (*day*, *night*) and identity contexts (*urgent*, *normal*) capture the variability in the time of the day and type of patient that needs attention. Location contexts (*corridor*, *entrance*, *pharmacy*, *medical-room*, and *care-center*) denote one or more physical locations on the map (A, B, C, D, E, F, G, WAITING, ISOLATION, and CHARGING). Both during the day and during the night, the robot must patrol the clinic. By patroling we mean that the robot should recurrently visit all the rooms (in any order) without letting any room unvisited. To express this property we use the patroling pattern [2].

During the day, and when inside the pharmacy, the robot must get the medicine, whenever asked to do so, and give it to the client. It must also welcome new patients at the entrance of the shop. Finally, it must always go and charge the battery when the power level is low. CROME relies on the model checker NuSMV [58] to perform all the checks in the CGG and on Strix[2] to generate controllers via reactive synthesis.

Figure 5 shows the mission requirements formalized as goals by the designer and the goals that are selected from the library of goals by CROME to refine the mission requirements. A goal is composed of (i) name, (ii) description, (iii) context, and (iv) objective (Section IV). For brevity, in Figure 5, we have omitted the goal descriptions. For example the name of the first goal is `Patrolling`, the context is described by two atomic propositions, `night` and `day`, and the objective is an instantiation of the patroling[3] pattern (i.e., instantiated on the AP $\mathtt{care-center}$). We use multiple Boolean propositions separated by comma to denote multiple goals, each instantiated in a single context. Specifying *Goal 1* with context $night, day$ is then equivalent to specifying two separate goals for $night$ and day, respectively. When the context is omitted from a goal, then we imply that the goal objectives must hold in all contexts. We also note that *Goal 4* instantiates a property specification pattern, the recurrent pattern,[4] which also uses the scope `between Q and R`.

We observe that *Goal 1* and *Goal 3* are refined differently according to the contexts specified by the designer. In *Goal 1*, patroling is achieved by patroling the *corridor* during the night, while patroling the *entrance* and the *pharmacy* are required during the day. Similarly, for *Goal 3*, the task of welcoming a new patient is refined differently according to the gravity of the symptoms where, in the *severe* cases, the robot visits the isolation room while, in the *normal* cases, it goes in the waiting room.

CROME produces a CGG with a total of 17 goals, 5 of which are the identified scenarios and 11 are library goals that can be reused to refine the leaves of the CGG. Figure 6 shows

[2]https://strix.model.in.tum.de/
[3]http://roboticpatterns.com/pattern/sequencedpatrolling/
[4]http://ps-patterns.wikidot.com/recurrence-property-pattern

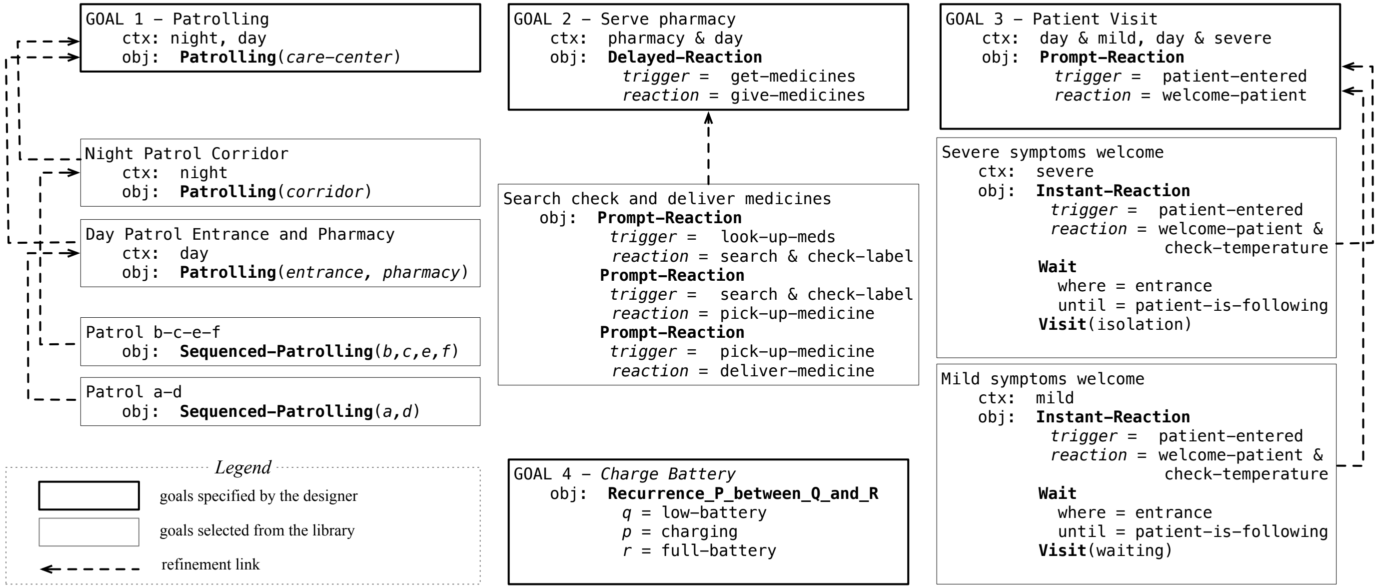

Fig. 5: Care clinic main goals specified by the designer connected via refinement to the goals selected from the library.

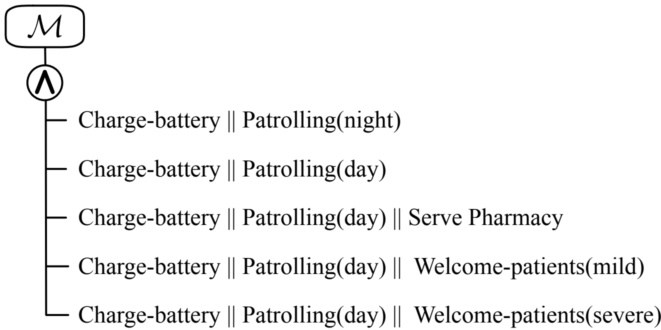

Fig. 6: Goals belonging to each identified scenario of the case study.

the name of the goals associated to each of the five mutex-contexts. The full example is available online.[5] The root node of the CGG can not be realized by the synthesizer, which runs out of memory. However, every scenario of the CGG is realizable with a maximum and minimum synthesis time of 962.98 s and 24.59 s, respectively. On the other hand, individual nodes can take up to 0.72 s to be realized into a controller. Overall, this example shows how CROME facilitates the formalization of mission requirements, can identify mission scenarios that can be independently realizable, and can refine leaf goals from a library of goals.

VIII. Conclusions

In this paper we introduced CROME, a design framework for capturing and formalizing robotic mission requirements. CROME facilitates the translation of informal requirements in terms of goals by leveraging a set of specification patterns. It then formalizes the goals in terms of assume-guarantee contracts and leverages a novel, modular representation, namely, a contract-based goal graph (CGG), to analyze the mission specification and detect inconsistencies. Given the CGG, CROME can automatically refine the leaf nodes with goals from a predefined library. It can automatically check the realizability of the overall mission and synthesize a controller, if the mission is realizable. Finally, if the mission is not realizable, CROME can help debug the specification by identifying subsets of requirements in terms of mission scenarios that may be realized. Future work includes devising heuristics to improve on the scalability of the clustering algorithm and further experimentation, also in collaboration with our industrial partners, to investigate the feasibility and usability of the approach in practical and industrial contexts.

[5]https://github.com/pierg/cogomo/tree/master/results/CROME

Acknowledgments

This work was supported in part by the Wallenberg AI Autonomous Systems and Software Program (WASP), funded by the Knut and Alice Wallenberg Foundation, and the EU H2020 Research and Innovation Program under GA No. 731869 (Co4Robots). In addition, the authors gratefully acknowledge the support by the US National Science Foundation (NSF) under Awards 1846524 and 1839842, the US Defense Advanced Projects Agency (DARPA) under Award HR00112010003, the US Office of Naval Research (ONR) under Award N00014-20-1-2258, Raytheon Technologies Corporation, and the Centre of EXcellence on Connected, Geo-Localized and Cybersecure Vehicle (EX-Emerge), funded by the Italian Government under CIPE resolution n. 70/2017 (Aug. 7, 2017). The views, opinions, or findings contained in this article should not be interpreted as representing the official views or policies, either expressed or implied, by the US Government. This content is approved for public release; distribution is unlimited.

References

[1] IFR, "World Robotic Survey," https://ifr.org/ifr-press-releases/news/world-robotics-survey-service-robots-are-conquering-the-world-, 2016.

[2] C. Menghi, C. Tsigkanos, P. Pelliccione, C. Ghezzi, and T. Berger, "Specification Patterns for Robotic Missions," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[3] Markets and Markets, "Service Robotics Market – Global Forecast to 2022," https://www.marketsandmarkets.com/Market-Reports/service-robotics-market-681.html, 2017.

[4] D. Bozhinoski, D. D. Ruscio], I. Malavolta, P. Pelliccione, and I. Crnkovic, "Safety for mobile robotic systems: A systematic mapping study from a software engineering perspective," *Journal of Systems and Software*, vol. 151, pp. 150 – 179, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121219300317

[5] ISO, "ISO - Robotics," https://www.iso.org/obp/ui/#iso:std:iso:8373:ed-2:v1:en, 2012.

[6] "Roomba Robot Vacuum Cleaners," https://www.irobot.se/roomba.

[7] S. García, P. Pelliccione, C. Menghi, T. Berger, and T. Bures, "High-level mission specification for multiple robots," in *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 127–140.

[8] ABB, "ABB makes robot programming more intuitive with Wizard Easy Programming software," https://shorturl.at/sFU15.

[9] S. Maoz and J. O. Ringert, "GR(1) synthesis for LTL specification patterns," in *Foundations of Software Engineering (FSE)*. ACM, 2015.

[10] M. Guo and D. V. Dimarogonas, "Multi-agent plan reconfiguration under local LTL specifications," *The International Journal of Robotics Research*, 2015.

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

[12] C. Menghi, S. Garcia, P. Pelliccione, and J. Tumova, "Multi-robot LTL Planning Under Uncertainty," in *Formal Methods*, K. Havelund, J. Peleska, B. Roscoe, and E. de Vink, Eds. Cham: Springer International Publishing, 2018, pp. 399–417.

[13] A. Ulusoy, S. L. Smith, X. C. Ding, C. Belta, and D. Rus, "Optimal multi-robot path planning with Temporal Logic constraints," in *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2011*. IEEE, 2011.

[14] G. E. Fainekos, A. Girard, H. Kress-Gazit, and G. J. Pappas, "Temporal logic motion planning for dynamic robots," *Automatica*, vol. 45, no. 2, pp. 343–352, 2009.

[15] M. Guo, K. H. Johansson, and D. V. Dimarogonas, "Revising motion planning under linear temporal logic specifications in partially known workspaces," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2013.

[16] E. M. Wolff, U. Topcu, and R. M. Murray, "Automaton-Guided Controller Synthesis for Nonlinear Systems with Temporal Logic," in *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2013.

[17] H. Kress-Gazit, "Robot challenges: Toward development of verication and synthesis techniques [errata]," *IEEE Robotics & Automation Magazine*, vol. 18, no. 4, pp. 108–109, 2011.

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

[19] S. Maoz and Y. Sa'ar, "AspectLTL: an aspect language for LTL specifications," in *International conference on Aspect-oriented software development*. ACM, 2011.

[20] S. Maoz and J. O. Ringert, "On well-separation of GR(1) specifications," in *Foundations of Software Engineering (FSE)*. ACM, 2016.

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

[22] X. Sun, R. Nambiar, M. Melhorn, Y. Shoukry, and P. Nuzzo, "DoS-resilient multi-robot temporal logic motion planning," in *Proc. International Conference on Robotics and Automation (ICRA)*, 2019, pp. 6051–6057.

[23] G. J. Holzmann, "The logic of bugs," in *Foundations of Software Engineering (FSE)*. ACM, 2002.

[24] M. Autili, P. Inverardi, and P. Pelliccione, "Graphical scenarios for specifying temporal properties: An automated approach," *Automated Software Engg.*, vol. 14, no. 3, 2007.

[25] W. Wei, K. Kim, and G. Fainekos, "Extended LTLvis motion planning interface," in *International Conference on Systems, Man, and Cybernetics*. IEEE, 2016.

[26] C. Lignos, V. Raman, C. Finucane, M. Marcus, and H. Kress-Gazit, "Provably correct reactive control from natural language," *Autonomous Robots*, vol. 38, no. 1, pp. 89–105, 2015.

[27] V. Raman, C. Lignos, C. Finucane, K. C. Lee, M. Marcus, and H. Kress-Gazit, "Sorry Dave, I'm Afraid I Can't Do That: Explaining Unachievable Robot Tasks Using Natural Language," University of Pennsylvania Philadelphia United States, Tech. Rep., 2013.

[28] S. Srinivas, R. Kermani, K. Kim, Y. Kobayashi, and G. Fainekos, "A graphical language for ltl motion and mission planning," in *International Conference on Robotics and Biomimetics (ROBIO)*. IEEE, 2013.

[29] U. S. Shah and D. C. Jinwala, "Resolving ambiguities in natural language software requirements: a comprehensive survey," *ACM SIGSOFT Software Engineering Notes*, 2015.

[30] N. Kiyavitskaya, N. Zeni, L. Mich, and D. M. Berry, "Requirements for tools for ambiguity identification and measurement in natural language requirements specifications," *Requirements engineering*, 2008.

[31] J. O. Ringert, B. Rumpe, and A. Wortmann, "A requirements modeling language for the component behavior of cyber physical robotics systems," *arXiv preprint arXiv:1409.0394*, 2014.

[32] A. Nordmann, N. Hochgeschwender, and S. Wrede, "A survey on domain-specific languages in robotics," in *Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2014.

[33] D. Bozhinoski, D. D. Ruscio, I. Malavolta, P. Pelliccione, and M. Tivoli, "FLYAQ: enabling non-expert users to specify and generate missions of autonomous multicopters," in *Automated Software Engineering (ASE)*. IEEE, 2015.

[34] D. Weintrop, A. Afzal, J. Salac, P. Francis, B. Li, D. C. Shepherd, and D. Franklin, "Evaluating CoBlox: A comparative study of robotics programming environments for adult novices," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, ser. CHI '18. New York, NY, USA: ACM, 2018, pp. 366:1–366:12.

[35] G. Biggs and B. Macdonald, "A survey of robot programming systems," in *in Proceedings of the Australasian Conference on Robotics and Automation, CSIRO*, 2003, p. 27.

[36] J. D. Robert W. Button, John Kamp, Thomas B. Curtin, *A Survey of Missions for Unmanned Undersea Vehicles*, 2010.

[37] A. Nordmann, N. Hochgeschwender, D. Wigand, and S. Wrede, "A Survey on Domain-Specific Modeling and Languages in Robotics," *Journal of Software Engineering for Robotics*, vol. 7, no. 1, pp. 75–99, 2016.

[38] SPARC, "Robotics 2020 Multi-Annual Roadmap," shorturl.at/rIQ07, 2016.

[39] S. Garcia, D. Struber, D. Brugali, T. Berger, and P. Pelliccione, "An empirical assessment of robotics software engineering," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*, 2020.

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

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

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

[43] C. Menghi, C. Tsigkanos, T. Berger, and P. Pelliccione, "Psalm: Specification of dependable robotic missions," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2019, pp. 99–102.

[44] P. Nuzzo, A. Sangiovanni-Vincentelli, D. Bresolin, L. Geretti, and T. Villa, "A platform-based design methodology with contracts and related tools for the design of cyber-physical systems," *Proc. IEEE*, vol. 103, no. 11, Nov. 2015.

[45] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, “Patterns in property specifications for finite-state verification,” in *Proceedings of the 21st international conference on Software engineering*, 1999, pp. 411–420.

[46] A. Van Lamsweerde, “Goal-oriented requirements engineering: A guided tour,” in *Proceedings fifth ieee international symposium on requirements engineering*. IEEE, 2001, pp. 249–262.

[47] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone *et al.*, “Contracts for system design,” *Foundations and Trends in Electronic Design Automation*, vol. 12, no. 2-3, pp. 124–400, 2018.

[48] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[49] S. Konrad and B. H. C. Cheng, “Real-time specification patterns,” in *Proc. of ICSE'05*. ACM, 2005, pp. 372–381.

[50] L. Grunske, “Specification patterns for probabilistic quality properties,” in *30th International Conference on Software Engineering (ICSE08)*, W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds. ACM Press, 2008, pp. 31–40.

[51] M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, and A. Tang, “Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar,” *IEEE Transactions on Software Engineering*, vol. 41, no. 7, pp. 620–638, 2015.

[52] “Property Specification Patterns,” http://ps-patterns.wikidot.com/.

[53] J. Krumm, *Ubiquitous Computing Fundamentals*, 2010, vol. 53, no. 5.

[54] A. K. Dey, “Understanding and using context,” *Personal and ubiquitous computing*, vol. 5, no. 1, pp. 4–7, 2001.

[55] D. D. Bloisi, D. Nardi, F. Riccio, and F. Trapani, “Context in Robotics and Information Fusion,” pp. 675–699, 2016.

[56] A. Van Lamsweerde, “Requirements engineering in the year 00: a research perspective,” in *Proceedings of the 22nd international conference on Software engineering*, 2000, pp. 5–19.

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

[58] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, “The nuXmv symbolic model checker,” in *CAV*, 2014, pp. 334–342.