Automated Synthesis of Controllers for Search and Rescue from Temporal Logic Specifications

Clemens Wiltsche
May 11, 2014

MASTER THESIS
INSTITUTE FOR AUTOMATION
ETH ZURICH

Supervising Professor:
Prof. John Lygeros

Supervisors:
Dr. Federico A. Ramponi
Sean Summers
Abstract

In this thesis, the synthesis of correct-by-construction controllers for robots assisting in Search and Rescue (SAR) is considered. In recent years, the development of robots assisting in disaster mitigation in urban environments has been actively encouraged, since robots can be deployed in dangerous and hazardous areas where human SAR operations would not be possible.

In order to meet the reliability requirements in SAR, the specifications of the robots are stated in Linear Temporal Logic and synthesized into finite state machines that can be executed as controllers. The resulting controllers are purely discrete and maintain an ongoing interaction with their environment by changing their internal state according to the inputs they receive from sensors or other robots.

Since SAR robots have to cooperate in order to complete the required tasks, the synthesis of controllers that together achieve a common goal is considered. This distributed synthesis problem is provably undecidable, hence it cannot be solved in full generality, but a set of design principles is introduced in order to develop specialized synthesizable specifications. In particular, communication and cooperation are resolved by introducing a verified standardized communication protocol and preempting negotiations between robots.

The robots move on a graph on which we consider the search for stationary and moving targets. Searching for moving targets is cast into a game of cops and robbers, and specifications implementing a winning strategy are developed so that the number of robots required is minimized.

The viability of the methods is demonstrated by synthesizing controllers for robots performing search and rescue for stationary targets and searching for moving targets. It is shown that the controllers are guaranteed to achieve the common goal of finding and rescuing the targets.
## Contents

List of Abbreviations

1 Introduction
1.1 Robot-Assisted SAR Scenarios
1.1.1 USAR
1.1.2 WiSAR
1.2 Challenges in Robot-Assisted SAR
1.3 Previous Work

2 Theoretical Background
2.1 Reactive Systems
2.1.1 Fair Discrete Systems
2.1.2 Temporal Logic
2.1.3 Model Checking and Deductive Proof Systems
2.2 Synthesis
2.2.1 Game Theoretic Approach
2.2.2 Computational Considerations
2.2.3 Synthesis Software
2.3 Compositional Synthesis
2.3.1 Game Theoretical Approach
2.3.2 Computational Considerations
2.3.3 Architectures
2.3.4 Composition
2.3.5 Decomposition

3 Specification
3.1 Classification of Formulae
3.1.1 Safety and Liveness Properties
3.1.2 Derived Classes
3.1.3 Assumption/Guarantee Specifications
3.1.4 GR[1] Specifications
3.1.5 Stutter-Invariance
3.1.6 Specification Example
3.2 Specifying Distributed Systems
3.2.1 Composing Specifications
3.2.2 Communication
3.2.3 The Four-Phase Handshake Protocol
List of Abbreviations

A/G  Assumption-Guarantee
BNF  Backus-Naur Form
CTL  Computation Tree Logic
FDS  Fair Discrete System
FIFO First in first out
FSA  Finite State Automaton
GCS  Graph Clearing Sequence
GR[1] Generalized Reactivity[1]
JTLV Java-TLV
LTL  Linear Temporal Logic
SAR  Search And Rescue
TL   Temporal Logic
TLA  Temporal Logic of Actions
TLV  Temporal Verification System
USAR Urban Search And Rescue
WiSAR Wilderness Search and Rescue
Chapter 1

Introduction

Robot-assisted Search and Rescue (SAR) has received increased public attention after the 1995 Hanshin-Awaji Earthquake in Kobe, Japan, and the attacks in the same year on the Murrah Federal Building in Oklahoma City, USA [80]: The RoboCup-Rescue project, started in 1998 at several Japanese universities is one of the first reactions of robotics researchers to the aforementioned earthquake, encouraging the development of robots assisting in disaster mitigation [80].

The US Federal Emergency Management Agency gives the following definition:

Urban search-and-rescue (USAR) involves the location, rescue (extrication), and initial medical stabilization of victims trapped in confined spaces[1].

A recent survey by Goodrich et al. [31] extends the application of SAR operations from confined urban spaces:

Wilderness search and rescue (WiSAR) operations include finding and providing assistance to humans who are lost or injured in mountains, deserts, lakes, rivers, or other remote settings.

Thus, SAR entails searching for people that are in distress or in need of medical support and providing relief to their situation. In an abstract sense, SAR includes the searching of any kind of problematic object and alleviating the situation. This requires teams of trained personnel, a well-managed organizational structure and management of equipment logistics.

Motivating Robot-Assisted SAR

Consider the scenario of a burning building, or the aftermath of a devastating hurricane. Usually there is only limited information available about whether there were people in the affected area at the time of the incident and what the imminent dangers to the rescue personnel are. Deploying SAR teams exposes personnel to the dangers of the site, and rescue efforts might be slow and costly.

[1]http://www.fema.gov/emergency/usr/ accessed 19 Oct 2011
Supporting the SAR efforts by robots has several advantages. Robots can be deployed in hazardous areas where human operations would not be possible. A major issue during disasters is the physical exhaustion and lack of sleep of SAR personnel, resulting in an increased susceptibility to misjudgements. Robots do not suffer from this kind of fatigue, even though their power supply needs regular recharging. Also, the precision of the sensors and speed of robots can by far outmatch those of humans. This is usually partially alleviated by canine-supported or mounted SAR teams, with the obvious limitations in hazardous or physically demanding environments.

As SAR robots obtain higher levels of autonomy, it can also be expected that the number of human operators per robot can be decreased. For each SAR robot that was employed during the operations after the September 11 attacks on the World Trade Center in 2001, at least two or more operators were needed, even though the robots had only very basic reconnaissance capabilities. Current examples of autonomous vehicles with greater operational capabilities are deployed in military applications, and require even a larger number of operators, e.g. 180 for the General Atomics Predator (but here the degree of autonomy is partially limited due to legal constraints). Automated synthesis for a particular class of specifications can be done efficiently in polynomial time. This could even allow feasible on-the-fly synthesis of controllers if the configuration changes during a field operation, e.g. by attaching a new sensor or by changing the means of locomotion. In this case, it is merely required to add the appropriate rules to the specification, as highlighted in the discussion of incremental modification at the beginning of Section 3. However, the main limitation of synthesis approach is the large state space size of the resulting controllers. Hand-coded controllers are less prone to this problem (see the comparisons in Bloem et al. [11]), but cannot be scaled to arbitrary size and complexity.

Contributions

In this thesis, the development of controllers for autonomous SAR robots is investigated. In order to act autonomously, a robot must be enabled to make decisions about the actions it performs, based on its observations about the environment and the other robots.

We consider purely discrete controllers for SAR robots that maintain an ongoing interaction with their environment by changing their internal state according to the inputs they receive. The specification of such controllers consists of a set of rules or guarantees such as “each location in the search area is visited repeatedly,” while allowing assumptions about the environment to be made, such as “targets cannot move faster than 3 mph.” Such statements are be expressed in a formal high-level specification language which closely matches how humans would naturally express such properties.

---

2. The Economist, Drones and the Man, Jul 30th 2011, http://www.economist.com/node/21524876, accessed 28 Oct 2011.
Instead of merely verifying properties, we are concerned with synthesizing controllers that are correct by construction. Using such an approach shifts the validation effort to the specification. However, expressing the specifications in a high-level language can greatly simplify this process.

SAR is typically performed using several agents that have to coordinate and communicate. We therefore need to consider the synthesis of controllers that together achieve a common goal. This is called compositional (or distributed) synthesis, and is in general undecidable [76]. The compositional synthesis problem therefore cannot be solved in its full generality, but we introduce a set of design principles that help to develop appropriate specifications that can be synthesized. Communication and coordination are resolved by introducing a standardized control protocol and preempting negotiations between robots.

The viability of our methods is demonstrated by synthesizing controllers for robots performing search and rescue for stationary targets and searching for moving targets. A four-phase handshake protocol is used to communicate commands between the agents. When interpreting the controllers in real-time, our main motivation is not to attain temporal efficiency, but to develop controllers that are proven to be reliable. We show that the controllers are guaranteed to achieve the common goal of finding and rescuing the targets and provide simulation results demonstrating the validity of the controllers.

1.1 Robot-Assisted SAR Scenarios

In this section, we consider two paradigms for robot-assisted SAR and outline the scenarios. The stationary target search in USAR and the moving target search in WiSAR guide our development of the specifications that are synthesized to be executed as controllers on SAR robots.

1.1.1 USAR

Robots assisting in USAR can be deployed for a variety of tasks, like inspecting damaged or collapsed buildings for imminent danger to human rescuers. Robots can also be used to monitor the temperature and air quality in a burning building or provide comfort to victims by establishing a two-way audio connection to rescue personnel. Casper and Murphy [15] provide a survey of such tasks related to the September 11, 2001 attacks on the World Trade Center.

We study controllers for USAR to develop an understanding of the tasks of searching and rescuing when a central coordinating authority exists. In USAR we can assume that injured or entrapped persons or objects are not able to move. It is therefore sufficient to develop methods for stationary target search, which considerably simplifies the controllers.
1.2. CHALLENGES IN ROBOT-ASSISTED SAR

1.1.2 WiSAR

While USAR is mostly concerned with small to medium scale sites such as burning or collapsed buildings, WiSAR entails searching over large regions in often rugged remote areas [31]. Robot-assisted WiSAR often combines several types of Unmanned Air- and Ground Vehicles in order to find a target that might be moving in order to get to safety itself.

The search strategies required for WiSAR are fundamentally different from those sufficient for USAR. This is mainly because the targets are assumed to be able to move. In this case, the number of robots required for reliable search depends on the topology of the environment. Also, the search radius has to increase with time, since a target might try to move away from the location at which it is initially expected to be, trying to reach safety itself. In this thesis we concentrate on developing a reliable distributed search strategy for a constant search radius.

1.2 Challenges in Robot-Assisted SAR

There are several challenges in developing and deploying SAR robots. The most critical ones are summarized below.

Unknown Environment Topology

In a typical SAR scenario, little is known about the topology of the environment in which the robots are operating. In USAR, even if building plans are available to a robot, passages and corridors might be blocked, voids might be opened and in a collapsed building the entire structure might be changed. Also, in a WiSAR application, the terrain might be unknown, and in particular, accurate elevation data is not always available. This is exacerbated by the need to expand the search radius as time progresses.

Searching an unknown environment can be accomplished by first obtaining an internal representation of the topology in which the robot is operating. The main drawback of this approach is that time is wasted on building this internal map that could be dedicated to rescuing. An alternative is to learn the map of the environment concurrently while searching [57, 58]. These approaches obtain a probabilistic representation of the environment and are therefore not applicable to the controller framework considered in this thesis.

Changing Environment Topology

Even if the robot has an accurate and reliable representation of the environment topology at some point in time, in a real-world scenario this knowledge has to be updated continually. An example of an application in which this capability is necessary is USAR in a burning building in which hallways and staircases might collapse. Another example is WiSAR after natural disasters such as landslides, floods or earthquakes in which the landscape keeps changing. Thus SAR robots
need a way of detecting the environment topology and adapting their movements through the SAR site accordingly.

This adds another layer of complexity to the task of synthesizing controllers that are correct by construction, since it has to be taken into account that the “knowledge” a robot has at each point in time might be changing. Synthesis guaranteeing such epistemic (knowledge-based) properties has been considered before [81] but not in the input-output driven way that is required for controllers.

Due to the problems with unknown and changing environment topologies, in this thesis we assume a constant environment and that an accurate and reliable representation of the environment topology is known at specification-time, i.e. when the specification is written for the SAR robots.

Cooperation and Communication

Often a SAR task requires the cooperation of many agents. For example, finding a moving target can be done reliably by a single robot only in a very limited number of cases, see Section 4.3. In order to cooperate, agents must be able to share relevant information about their own actions and adapt their strategies accordingly.

Communication between robots is governed by protocols which must be encoded in the specification. However, this introduces circularity: A robot has to guarantee to satisfy its part of the protocol’s contract only if the robot with which it communicates satisfies its own part. A naïve formalization of this principle would allow the robots to exhibit any behavior in the case that no robot satisfies its guarantees initially. Resolving these circularities requires to impose restrictions on the specifications pertaining to communication, giving rise to assumption/guarantee specifications [36] and the corresponding composition rules, see Section 3.2.

Realizability and Synthesizability

Synthesis of correct-by-construction controllers has the obvious advantage that the specification is guaranteed to be satisfied. However, there exist specifications that cannot be realized (or satisfied) by any controller. Thus, if the specification is unrealizable, no controller can be generated from it, independently of the synthesis method used.

Not all specifications that are realizable can also be synthesized into a controller, depending on the synthesis method used. When choosing the synthesis method, there is a tradeoff between the generality of the specifications supported and the efficiency of the synthesis and of the resulting controllers.

The realizability of some specifications depends on the topology of the environment. For example, the property “each location in the search area is visited repeatedly” requires that the topology can be represented by a strongly
1.3. PREVIOUS WORK

connected graph, see Section 4.1. Deciding realizability and synthesizing controllers for a particular topology can be handled reasonably well by existing methods. However, controllers that provide guarantees for all graphs classified by some user-specified properties (such as connectedness or Hilbertness) pose a much harder problem and is not treated here.

Integration and Human-Robot Interaction

One of the main challenges in developing robots assisting SAR is that they have to be integrated in existing organizational structures. This includes technical aspects such as the tasks that a robot is capable to perform, and the improvements that this yields over not deploying the robots alongside human SAR personnel. However, legal considerations such as liability in the case of failure or lack of trust of team leaders in the robots’ capabilities may prevent SAR robots from being deployed even when there are no technical difficulties or limitations.

Moreover, the interaction of humans with the robots must be taken into account. One major concern is the need for trained operators that must dispatch, control and maintain the robots. Also, automatic detection of victims is not yet reliable enough for completely autonomous SAR, so human operators are required to interpret the sensor data [15]. While those aspects mainly concern the search for the victims, automated on-site rescue and medical treatment poses a variety of new challenges.

1.3 Previous Work

The previously mentioned RoboCup-Rescue project initiated a surge of research in robot-assisted SAR. However, even as early as 1984, computer assistance to SAR has been considered by Belardo et al. [8]. Early attempts to robot-assisted SAR include supportive technologies for robot deployment [10], coordinating robots to perform rescue-like tasks [5, 55] and combining this with cooperative search [35].

The RoboCup-Rescue project and other similar projects simulate USAR scenarios in which several robot-developing teams can test their devices and hold competitions. With the advancements from such early research projects, several SAR robots were deployed during the World Trade Center rescue response in 2001 [15]. While robot deployment time was limited, this unfortunate incident provided valuable information for further development for SAR robots.[3]

Recent research into robots assisting in tasks connected with SAR has demonstrated several successful implementations. Several types of robots such as tracked, legged and flying vehicles, snakes and even shape-shifters have been
1.3. PREVIOUS WORK

developed that allow access into damaged buildings, hazardous areas and
confined spaces [19, 25, 34, 63]. Some of these robots have already been deployed
in actual SAR tasks [15, 51, 62].

Instead of focusing on the physical properties of the SAR robots, this thesis is
predominantly concerned with the control of teams of robots. This has been
considered previously, e.g. for building inspection and searching [37, 88]. However,	en these teams need to perform complex tasks that can be described
quite straightforwardly in a rule-based way, but are hard to implement manually
into a discrete controller that obeys all the rules at the same time. A promising
approach is to automatically synthesize correct-by-construction controllers from
high-level descriptions [47].

The synthesis of correct-by-construction controllers has been considered since
the 1960’s for example by Church, Büchi, Landweber, and Rabin. Initially,
the results were developed for very general properties, leading to a pessimistic
outlook on the computational complexity of synthesis. Also, only recently have
methods been developed that are directly applicable for controller synthesis [71].
This seminal work by Piterman et al. enabled efficient synthesis of controllers
from temporal logic specifications and forms the basis of synthesis tools devel-
oped at the California Institute of Technology and the University of Pennsylva-
nia [47, 91]. The respective research groups augment the discrete controllers re-
sulting from the temporal logic specifications with continuous controllers. Other
approach to synthesizing such hybrid controllers that are not based on the work
by Piterman et al. were developed by Kloetzer and Belta [40] and Loizou and
Kyriakopoulos [55]. An overview of current techniques is given in the work by
Belta et al., which goes beyond the techniques covered in this thesis [9].

Several successful examples and implementations of controllers that have been
synthesized from high level specifications have been developed [27, 58, 89]. Toy
examples of robots moving in buildings and completing tasks include a conceptu-
tual implementation of USAR [46]. Controller synthesis has also been suggested
to be used for vehicles competing in the DARPA Grand Challenge but for such a
large-scale application several hurdles have to be overcome first [14]. Still, mod-
ing and synthesizing controllers from high-level specifications for symbolic
control and planning is becoming increasingly powerful [9], and several software
front-ends for synthesis of discrete and hybrid systems such as TuLiP [91], LTL-
Con [43], LTLmop [27] and PESSOA [59] have been developed since.
1.3. PREVIOUS WORK
Chapter 2

Theoretical Background

We consider the synthesis of controllers for SAR robots that are guaranteed to satisfy their specifications by construction. Besides the synthesis procedure, a formal description language of the resulting controllers as well as a formal specification language have to be introduced.

In Section 2.1 we introduce an abstract model of discrete reactive controllers and how to reason about their properties. The synthesis procedure of a single correct-by-construction controller from its specification is outlined in Section 2.2. Since several agents cooperate in an SAR scenario, compositional synthesis is introduced in Section 2.3.

2.1 Reactive Systems

A controller must maintain a continuous interaction with its environment and the system it controls. The system can be controlled by changing set points of actuators, but the environment may only be sensed and not influenced by the controller. The system keeps a state that is changed according to the behavior of its controller and its environment. The controller maintains its knowledge about the system’s state and thus the controller is simply considered to be part of the system.

A traditional computer program can be regarded as relation between a set of initial states and a set of final states, acting as state transformers. However, this paradigm is not appropriate for describing the program of a controller, since it has to refer to the controller’s ongoing behavior. A controller reacts to changes in its inputs and its program is therefore called a reactive system. The behavior of a reactive system can be described by a (possibly infinite) sequence of states of the system. This sequence can be influenced by the environment via events or shared variables, which can be modelled by incorporating appropriate atomic propositions into the specification language.

---

1In traditional control terminology the system is considered to be fully observable.
We consider a controller as a reactive system in discrete time. Since in this thesis the synthesized controllers are simulated as if they were running directly on digital microprocessors, this temporal abstraction is sensible. Moreover, we are mainly concerned with specifying and synthesizing controllers that decide on a robot’s high-level moves, similar to the moves in a game of chess, rather than the robot’s continuous trajectories.

### 2.1. Fair Discrete Systems

A reactive system can be represented as a Finite State Automaton (FSA) that captures how the system state evolves over time. Piterman et al. suggest the use of a Fair Discrete System (FDS) as an abstract computational model for reactive systems [71]. We make the inputs, the outputs and the state space explicit, as it will later simplify the description of systems consisting of several FDS’s.

**Definition 1.** A Fair Discrete System $M$ is a tuple $(V, \mathcal{X}, \mathcal{Y}, Q, \Theta, \rho)$ consisting of the following components:

- $V = \{u_1, u_2, \ldots, u_n\}$ is a finite set of typed state variables, over finite domains $\{\mathcal{V}(u_1), \mathcal{V}(u_2), \ldots, \mathcal{V}(u_n)\}$ respectively. Let $\mathcal{V}(V) = \prod_{u \in V} \mathcal{V}(u)$ denote the set of all possible valuations of all variables in $V$, called the alphabet of $M$.
- $\mathcal{X} \subseteq V$ is the set of environment variables that are controlled by the environment. They are also called inputs.
- $\mathcal{Y} = V \setminus \mathcal{X}$ is the set of system variables that are controlled by the system (i.e. the actuator set points). They are also called outputs.
- $Q \subseteq \mathcal{V}(V) \times \mathbb{N}_0$ is the set of states. A state $q \in Q$ is a valuation with a unique identifier, so there can be two distinct states corresponding to the same valuation. We write $q[u]$ to denote the value of the variable $u \in V$ in the state $q$, i.e. $q[u] \in \mathcal{V}(u)$. A state $q$ satisfies a propositional formula $\varphi$, written $q \models \varphi$ iff $\varphi$ holds when for each occurrence of $u$, the corresponding value $q[u] \in \mathcal{V}(u)$ is substituted.
- $\Theta$ is the initial condition, a propositional formula characterizing the initial states of $M$. A state $q \in Q$ is initial iff $q \models \Theta$.
- $\rho$ is the transition relation defining the behavior of the FDS. It is a function $\rho : Q \rightarrow 2^Q$ that takes the current state $q \in Q$ and produces a set of new states $Q' \subseteq Q$. Note that this formulation allows for nondeterministic transition relations by producing sets containing two or more states. If $q' \in \rho(q)$ this is also written as $q \rightarrow_{\rho} q'$.

An FDS describes how the values of the variables in $V$ change over discrete time. The concept of how time evolves and relates to real time is described in detail in Section 2.1.2. As an FDS proceeds from state to state, a sequence over $Q$ is generated. In order to reason about the behavior of an FDS, we introduce the concept of a computation:

---

2Strictly, we introduce what is called fairness-free FDS’s. Fairness requires that certain transitions are taken infinitely many times [73].
Definition 2. A computation of an FDS $M = (V, X, Y, Q, \Theta, \rho)$ is a (possibly infinite) sequence of states $q = q_0q_1q_2\ldots$ over $Q$ satisfying

(C1) Initiality: $q_0 \models \Theta$.

(C2) Consecution: $\forall j \in \mathbb{N}. j \leq |q| \Rightarrow q_{j-1} \rightarrow_{\rho} q_j$.

Here $|q|$ denotes the length of the sequence $q$: it is $\aleph_0$ if the sequence is infinite, and $n$ for a finite sequence $q = q_0q_1\ldots q_n$. Thus, the length of a computation is the number of transitions made. We also introduce the concatenation of two sequences $\sigma_1 = s_0s_1\ldots s_n$ and $\sigma_2 = s_{n+1}s_{n+2}s_{n+3}\ldots$, where $\sigma_1$ must be finite, to be the sequence $\sigma_1\sigma_2 = s_0s_1s_2\ldots$, i.e. concatenations of sequences are written just by concatenating the symbols.

We are only interested in reactive systems that maintain an ongoing interaction with their environment, hence only infinite computations will be considered. FDS’s for which all maximal length computations are infinite are called non-blocking and are characterized by the additional requirement that in each state the transition relation allows the FDS to enter a next state:

(NB) Nonblocking Property: $\forall q \in Q. \rho(q) \neq \emptyset$.

Moreover, any physical system has an initial state. To reflect this in our definition of an FDS, the following property is added:

(SP) Startup Property: $\exists q_0 \in Q. q_0 \models \Theta$.

Definition 3. Define the (overloaded) operator $\mathcal{V}_W : Q \rightarrow \mathcal{V}(W)$ to map states to their corresponding valuations. That is, for a state $q = (v, n) \in Q$, $\mathcal{V}_W(q) = v \in \mathcal{V}(W)$. If $W$ in $\mathcal{V}_W$ is understood from context, we simply write $\mathcal{V}$.

The $\mathcal{V}$-operator can also be used for sets, sequences and sets of sequences of states, mapping each element to its corresponding valuation. Moreover, $\mathcal{V}$ is used to map variables to their domains, so it is “overloaded” to yield the domain $\mathcal{V}(u)$ for a variable $u$. The sets of valuations of sets of variables $V$, $X$ and $Y$ are also defined using $\mathcal{V}$ as $\mathcal{V}(V)$, $\mathcal{V}(X)$ and $\mathcal{V}(Y)$.

Definition 4. The set of all finite nonempty sequences over a set $S$ is denoted by $S^+$, and the set of all infinite sequences over $S$ is denoted by $S^\infty$. 

Note that by (SP) all computations of an FDS $M$ are nonempty, as $M$ must have an initial state satisfying (C1). The set of infinite computations of an FDS $M$ is then denoted by $\mathcal{C}(M) \subseteq Q^\infty$, and the corresponding set of infinite sequences of valuations, called the language of $M$ is $\mathcal{V}(\mathcal{C}(M)) \subseteq \mathcal{V}(V)^\infty$. To simplify notation, we denote the language of $M$ by $\mathcal{L}_M = \mathcal{V}(\mathcal{C}(M))$.

A transition of an FDS can either be triggered by a change of the environment variables, called environment-triggered, or by some external events such as the ticks of a system clock on a computer, called clock-triggered. In a purely environment-triggered FDS, no transitions can be made that merely change the system variables if no changes in the inputs are observed, limiting the set
of behaviors (the language) expressible by such FDS’s. However, in a purely
clock-triggered FDS, changes in the environment variables might be overlooked
if several such changes occur during one tick of the clock. Since the FDS’s that
we consider should be implementable on a microprocessor, we adopt a clock-
triggered approach and account for the complications in the specifications, see
in particular Section 3.1.5 and Section 3.2.2.

If \( \mathcal{X} \) is nonempty, a change in the any of the environment variables in
\( \mathcal{X} \) requires a transition to be made. Since \( \mathcal{X} \) is not controlled by the FDS, the nonblock-
ing condition (NB) is no longer sufficient to ensure that there is always a valid
transition available. The stronger requirement that in every state a transition
is possible independently of the inputs is sufficient to ensure that an FDS al-
ways maintains an ongoing interaction with its environment. However, often
this requirement is too strong for the specification to be realizable. This issue is
addressed by introducing assumption/guarantee specifications in Section 3.1.3.

By moving from state to state, an FDS can represent the behavior of a micro-
processor. While an FDS can be considered as the “program” running on the
microprocessor, it abstracts away from any specific hardware model. As long as
a microprocessor implements the behavior of an FDS, it is sufficient to reason
about the FDS — the results will hold for the actual implementation as well.
Indeed, we will later show how to simulate an FDS and use this to test the
synthesized implementations.

While the behavior of reactive systems can be accurately modeled using FDS’s,
only very basic reactive systems can be hand-coded in a manageable amount
of time. In order to handle large systems, a high-level system specification lan-
guage is used to describe the operation of an FDS. Synthesis from a correct
high-level description yields a correct-by-construction controller in the form of
an FDS. Correctness of a system can then be methodically verified in the speci-
fication language without having to explicitly reason about its semantics in the
foundation logic.

2.1.2 Temporal Logic

Temporal Logic (TL) is a well-established formalism that can be traced back
even to the ancient Greeks [32]. It has been formalized as Tense Logic by Prior
in 1962 [70] and introduced as a formal specification language for reactive sys-
tems by Pnueli in 1977 [72]. A statement in propositional logic is interpreted
to make a statement over the state at one given time (the present). However,
with TL also statements about the past and the future can be made.

The Concept of Time

The natural concept of time is that it proceeds continuously and at a constant
rate. In temporal logic, this concept of real-time is abstracted away in two ways.
Firstly, time in TL proceeds in discrete steps, that might be aligned to clock
ticks or other external events. The present is considered to be at time zero. A
propositional formula $p$ just talks about the present:

\[ t \rightarrow p \]

However, with a temporal formula, statements about several points in time can
be made. For example, with the “always” operator, $\Box p$ we can express that $p$
is an invariant over time (in the future):

\[ t \rightarrow p \cup p \cup p \cup p \cup \ldots \]

And with the “eventually” operator, $\Diamond p$ it can be expressed that the proposition
$p$ will hold at some unspecified future point in time:

\[ t \rightarrow p \]

Other temporal operators express similar properties. Several forms of TL are
in use, but here we only consider Linear Temporal Logic (LTL). It is called
“linear” because LTL formulae are interpreted over sequences of states, where
each point in time is associated with an element in the sequence. Thus for a
sequence of states $s_0s_1s_2\ldots$, the element $s_0$ is said to occur at time zero, $s_1$ is
said to occur at time one and so on. These times can be the ticks of a clock.
However, there is no correspondence to a real-time axis, which is the second
abstraction step. Thus only the order and perhaps the number of repetitions of
states in a sequence has semantic significance.

After discretizing and abstracting away from real time, the “past”, “present”
and “future” get their meaning from the part of the sequence under consider-
ation. A propositional formula talks about a particular instance in time, a
past formula talks about (non-strictly) smaller time indices, and a future
formula talks about (non-strictly) greater time indices. Time moves strictly
forward, so the elements of a sequence are considered one after the other, with-
out repetitions.

By interpreting the abstract time in our real-world notion of continuous time,
it can be defined what it means to execute an FDS on a microprocessor. The
processor clock generates a sequence of ticks $t_0t_1t_2\ldots$ that trigger the transi-
tions of an FDS. A clock-triggered FDS $\mathcal{M}$ is executed by starting in an initial
state $q_0=\Theta$ at clock tick $t_0$. Then, before every clock tick $t_i$, $i > 0$, the current
values $x_{i-1} \in V(\mathcal{X})$ of the input variables $\mathcal{X}$ are read from the environment’s
state and a transition is made from $q_{i-1}$ to $q_i$, s.t. $V(q_i)[\mathcal{X}] = x_{i-1}$, where $[\mathcal{X}]$
denotes the restriction to the variables in $\mathcal{X}$. Note that the current values of
the environment are only visible in the next state.
2.1. REACTIVE SYSTEMS

| Operator(s) | Precedence | Associativity |
|-------------|------------|---------------|
| ¬, ○, □, ♦, ■, ◆ | 1          | right         |
| U, U        | 2          | right         |
| ⊗           | 3          | right         |
| ∧           | 4          | left          |
| ∨           | 5          | left          |
| →, ↔        | 6          | right         |

Table 2.1: Operator precedence and associativity.

Syntax

The syntax of an LTL formula $\varphi$ over the variables $V$ is given by the BNF definition

$$\varphi ::= p \mid \neg \varphi \mid \varphi \lor \psi \mid \Diamond \varphi \mid \varphi U \psi,$$

where $p$ is an atomic proposition relating a variable $u \in V$ to a subset of its domain, $v \subseteq \mathcal{V}(u)$. That is, $p = (u, v) \in V \times \mathcal{V}(u)$. An atomic proposition of the form $p = (u, \{v \in \mathcal{V}(u) \mid v \bowtie w\})$ is written simply as $u \bowtie v$, where $w \in \mathcal{V}(u)$ is a constant and $\bowtie$ is one of $=, \neq, \leq, <, \geq$ or $>$. For a boolean variable $u$, the atomic propositions "$u = True$" and "$u = False$" are abbreviated simply by $u$ and $\neg u$ respectively.

The operator $\Diamond$ is called weak next or just next and $U$ is called strong until or just until. Next and until are called temporal operators, while $\neg$ and $\lor$ are called propositional operators. Other propositional operators can be defined in the usual way:

$$
\begin{align*}
\text{True} & \triangleq \varphi \lor \neg \varphi \\
\varphi \land \psi & \triangleq \neg (\neg \varphi \lor \neg \psi) \\
\varphi \rightarrow \psi & \triangleq \neg \varphi \lor \psi \\
\varphi \leftrightarrow \psi & \triangleq \varphi \rightarrow \psi \land \psi \rightarrow \varphi
\end{align*}
$$

A formula only containing atoms and propositional operators is called a propositional formula. The following additional temporal operators can be defined:

$$
\begin{align*}
\Diamond \varphi & \triangleq \text{True} U \varphi \\
\Box \varphi & \triangleq \neg \Diamond \neg \varphi \\
\varphi U \psi & \triangleq \varphi U \psi \lor \Box \varphi \\
\Diamond \varphi & \triangleq \neg \Box \neg \varphi
\end{align*}
$$

The operator $\Diamond$ is called eventually, $\Box$ is called always, $U$ is called weak until and $\circ$ is called strong next. Additionally, the temporal past operators $\blacksquare$ and $\spadesuit$ can be introduced, which will be used in Section 3.2.1. However, it can be shown that past operators do not add expressivity [53]. A formula that contains no past operators is called a future formula, while a formula that contains only propositional and past operators is called a past formula. Operator precedence and associativity are summarized in Table 2.1.
Semantics

Let \( \mathcal{L}(V) \) denote the set of syntactically correct LTL formulae over the variables \( V \), i.e. the language generated by \( \{2.1\} \). We now define a compositional semantics of LTL via the satisfaction relation \( \models \) between a tuple \((\sigma, j) \in V(V)^\infty \times \mathbb{N}_0 \) and an LTL formula \( \varphi \in \mathcal{L}(V) \). Since the semantics is defined over sequences of valuations \( V(V) \), LTL formulae are not tied to a particular FDS with its corresponding state space \( Q \). If two states \( q \) and \( q' \) correspond to the same valuation, i.e. \( V(q) = V(q') \), then there is no way to tell them apart in an LTL formula.

Given a finite or infinite sequence \( \sigma = s_0s_1s_2 \ldots \in V(V)^\infty \cup V(V)^+ \) and an index \( j \in \mathbb{N}_0 \), the relation \( \models \) is defined as follows:

\[
\begin{align*}
(\sigma, j) \models p & \iff s_j \models p \\
(\sigma, j) \models \neg \varphi & \iff (\sigma, j) \not\models \varphi \\
(\sigma, j) \models \varphi \lor \psi & \iff (\sigma, j) \models \varphi \quad \text{or} \quad (\sigma, j) \models \psi \\
(\sigma, j) \models \Box \varphi & \iff (\sigma, j + 1) \models \varphi \\
(\sigma, j) \models \varphi \U \psi & \iff \exists k \geq j. (\sigma, k) \models \psi \land (\forall i. j \leq i < k \Rightarrow (\sigma, i) \not\models \varphi).
\end{align*}
\]

The semantics of the past operators \( \Box \) and \( \Diamond \) is given by

\[
\begin{align*}
(\sigma, j) \models \Box \varphi & \iff \forall k. 0 \leq k \leq j \Rightarrow (\sigma, k) \models \varphi \\
(\sigma, j) \models \Diamond \varphi & \iff \exists k. 0 \leq k \leq j \land (\sigma, k) \models \varphi.
\end{align*}
\]

The formula \( \varphi \) is said to hold at position \( j \) of the sequence \( \sigma \in V(V)^\infty \) iff \( (\sigma, j) \models \varphi \).

In this semantics \( \Box \varphi \) means that if there is a next state, it satisfies \( \varphi \). In contrast, \( \Diamond \varphi \) means that there exists a next state and it satisfies \( \varphi \). If only infinite sequences are considered, these two operators are equivalent. In this thesis we will always use \( \Box \), since only nonblocking FDS’s are considered that can always react to a well-behaved environment. Note also that for infinite sequences \( \Box \) commutes with \( \neg \) and distributes over binary propositional operators.

Further, \( \varphi \U \psi \) means that \( \varphi \) holds until \( \psi \) holds, and \( \psi \) will eventually hold, justifying the name strong until. In contrast, \( \varphi \U \psi \) does not require \( \psi \) to eventually hold (i.e. \( \varphi \) holds forever) and is therefore called weak until.

A formula \( \varphi \in \mathcal{L}(V) \) is said to be satisfiable iff there exists a sequence \( \sigma \in V(V)^\infty \) and an index \( j \leq |\sigma| \) s.t. \( (\sigma, j) \models \varphi \) holds.

A sequence \( \sigma \in V(V)^\infty \) is said to satisfy a formula \( \varphi \in \mathcal{L}(V) \) iff for all indices \( j \leq |\sigma| \), \( (\sigma, j) \models \varphi \) holds. It is said to initially satisfy \( \varphi \) iff \( (\sigma, 0) \models \varphi \) holds. The latter is written \( \sigma \models \varphi \).

An FDS \( M \) is said to (initially) satisfy a formula \( \varphi \in \mathcal{L}(V) \) iff all sequences \( \sigma \in \mathcal{L}_M \) (initially) satisfy \( \varphi \). A formula \( \varphi \in \mathcal{L}(V) \) is called (initially) valid for an FDS \( M \) iff every sequence \( \sigma \in \mathcal{L}_M \) (initially) satisfies \( \varphi \). If \( M \) initially satisfies \( \varphi \), or \( \varphi \) is initially valid for \( M \), then this is written as \( M \models \varphi \).

15
2.1. REACTIVE SYSTEMS

A formula $\phi \in \mathcal{L}(V)$ is said to be (initially) valid iff all sequences $\sigma \in V(V)\infty$ (initially) satisfy $\phi$. If $\phi$ is initially valid, then this is written as $\vDash \phi$.

An LTL formula thus can be seen as specifying a set of sequences that initially satisfy it. LTL is called “linear” because it can only make statements about paths along the transitions in an FDS.

Other Temporal Logics in Use

In contrast to the linear-time LTL, branching time logics such as CTL can make statements about entire transition trees in an FDS [21]. in particular, this includes nondeterministic choices. There are properties that can be expressed in CTL but not in LTL and vice versa. Both specification languages are subsumed in a more general temporal logic called CTL* [22].

Several other logics for specifying reactive systems exist. Some are tailored for hardware, others for software. Interval Temporal Logic makes reasoning about periods of time explicit [61]. Atomic propositions in LTL make statements about variables at one point in time, but in the Temporal Logic of Actions introduced by Lamport in 1994 atomic propositions can be made over changes in variables that are termed “actions” [51].

2.1.3 Model Checking and Deductive Proof Systems

In order to make use of specifications, it must be possible to determine whether a reactive system, in form of an FDS $\mathcal{M}$, satisfies a given specification $\phi$. Model checking determines whether $\phi$ is (initially) satisfied by $\mathcal{M}$. The outline of this process is shown in Figure 2.1.

A formula $\varphi \in \mathcal{L}(V)$ is said to be (initially) valid iff all sequences $\sigma \in V(V)\infty$ (initially) satisfy $\varphi$. If $\varphi$ is initially valid, then this is written as $\vDash \varphi$.

An LTL formula thus can be seen as specifying a set of sequences that initially satisfy it. LTL is called “linear” because it can only make statements about paths along the transitions in an FDS.

Other Temporal Logics in Use

In contrast to the linear-time LTL, branching time logics such as CTL can make statements about entire transition trees in an FDS [21]. in particular, this includes nondeterministic choices. There are properties that can be expressed in CTL but not in LTL and vice versa. Both specification languages are subsumed in a more general temporal logic called CTL* [22].

Several other logics for specifying reactive systems exist. Some are tailored for hardware, others for software. Interval Temporal Logic makes reasoning about periods of time explicit [61]. Atomic propositions in LTL make statements about variables at one point in time, but in the Temporal Logic of Actions introduced by Lamport in 1994 atomic propositions can be made over changes in variables that are termed “actions” [51].

2.1.3 Model Checking and Deductive Proof Systems

In order to make use of specifications, it must be possible to determine whether a reactive system, in form of an FDS $\mathcal{M}$, satisfies a given specification $\phi$. Model checking determines whether $\phi$ is (initially) satisfied by $\mathcal{M}$. The outline of this process is shown in Figure 2.1.
Moreover, it is often useful to prove properties about a specification, without even considering the reactive system it will be applied to. Deductive proof systems require an LTL specification $\varphi$ to be supplied and will attempt to derive whether $\models \varphi$ holds independently of a particular FDS, i.e. $\forall \mathcal{M}. \mathcal{M} \models \varphi$. This process is outlined in Figure 2.2.

Model Checking

In order to check the (initial) validity of an LTL formula $\varphi$ over a fixed FDS $\mathcal{M}$, a model checker can be employed. The model checker in Figure 2.1 is provided with $\varphi$ and $\mathcal{M}$ and returns true if all sequences $\sigma \in \mathcal{L}_\mathcal{M}$ in the language of $\mathcal{M}$, (initially) satisfy the specification $\varphi$. The techniques and algorithms for model checking are manifold and can be found for example in \cite{18}.

Figure 2.3 shows an FDS $\mathcal{M} = (V, \mathcal{X}, \mathcal{Y}, Q, \Theta, \rho)$ with three variables and five states. Each state is represented as a circle, with the valuations of the state variables $V = \{x, y, z\}$ shown (the unique indices of the states are omitted). In this example, the partition of $V$ into $\mathcal{X}$ and $\mathcal{Y}$ is not relevant, and we may (rather arbitrarily) choose to have $\mathcal{X} = \emptyset$ and $\mathcal{Y} = V$. The arrows represent the transition relation: There is an arrow from a state $q$ to a state $q'$ iff $q \rightarrow_\rho q'$. There is an arrow without a source to a state $q$ iff $q \models \Theta$. Then, model checking could for example establish that the following specifications hold for this FDS:

$$\square y,$$
$$\square \Diamond \lnot x,$$
$$\neg x \land \neg z \rightarrow \square (\neg x \land y).$$

Figure 2.3: Example of an FDS with three variables and five states.

\footnote{Assuming soundness of the model checker.}
2.2. SYNTHESIS

Model checking can also be useful when synthesizing an FDS from its specification. First, the correctness of the synthesis can be verified. Also, if the synthesis algorithm does not support a particular specification (e.g., the until operator, $\mathcal{U}$ is not supported by the synthesis procedure we are using), then it can still be verified post-hoc whether the unsynthesizable properties are satisfied by the resulting FDS, even though not taken into account explicitly during synthesis.

Sequent Calculus for LTL

Sequent calculus is a deductive proof system that allows to prove tautologies in the specification language independent of a particular FDS. Proofs can even be automated up to a certain extent, e.g., using tools such as Isabelle [64].

This can be employed for example to prove the equivalence of LTL formulae, allowing for simplifications and translations between syntactic forms. Also, when modeling assumptions on the environment, it may be proved that they are satisfiable, avoiding the “false implies everything” problem. However, the satisfiability problem for the fragment of LTL that we consider is PSPACE complete, and is NP complete for the fragment of LTL supported by the synthesis procedure that we use (84, Figure 3, see the entries for $L(F, X)$ and $\tilde{L}(F, X)$ respectively).

Sequent calculus for LTL allows to derive temporal formulae by applying axioms and transformation rules [32]. A temporal formula $\varphi$ that can be derived in such a way is called a theorem, written $\vdash \varphi$. The proof system consisting of axiomatic sequents and rules should be sound, meaning that it is only possible to derive valid LTL formulae. Thus the soundness of a proof system can be expressed by the implication $\vdash \varphi \Rightarrow \models \varphi$. It is also desirable that the proof system is complete, meaning that it is possible to derive all valid formulae. This can be expressed by the implication $\models \varphi \Rightarrow \vdash \varphi$.

Theorems can be derived by purely syntactic means using sequent calculus [12, 29, 32]. This establishes a set of rules to derive theorems. Usually these rules are applied backwards, starting at the theorem to be proved and working towards axioms such as $\vdash True$.

The modus ponens is an example of a sequent calculus rule:

$$\vdash p \quad \vdash p \Rightarrow q \quad \vdash q.$$  

This rule says that $q$ can be derived by first deriving $p \Rightarrow q$ and $p$ separately. Both $p$ and $p \Rightarrow q$ must be similarly derived by rules in the sequent calculus, until one arrives at axioms such as the already presented one.

2.2 Synthesis

Synthesis is the automatic construction of programs and (digital) designs from logical specifications. It has been investigated already since the 1960’s for example by Church, Büchi, Landweber and Rabin. The choice of the synthesis
2.2. SYNTHESIS

algorithms depend on which specification language is used. For example, synthesis of open reactive systems from LTL has been considered by Pnueli and Rosner in 1989 [75], and again by Piterman et al. in 2006 [71] and Klein and Pnueli in 2011 [39].

Other specification languages have been considered. So far, synthesis from branching time logic is impractical, as the general problems of synthesis from CTL and CTL* specifications are in the complexity classes 2EXPTIME and 3EXPTIME\(^4\) respectively [49], and to our knowledge, no methods are available that perform better on a reduced set of specifications, although an interesting approach is suggested by Antoniotti and Mishra [4]. Synthesis of open controllers from Probabilistic CTL [3, 7] is in NP \(\cap\) co-NP [16].

We consider the synthesis problem to be to find an FDS \(M\) that initially satisfies a given LTL specification \(\varphi\), i.e. \(\forall \sigma \in \mathcal{L}_M. \sigma \models \varphi\). This is shown in Figure 2.4 where the synthesizer is only provided with the specification.

Typically, synthesis is preceded by deciding realizability of the LTL specification, which amounts to checking whether there exists an FDS \(M\) initially satisfying \(\varphi\), see Figure 2.5.

![Figure 2.4: Outline of Synthesizer.](image1)

![Figure 2.5: Outline of Realizability Check.](image2)

2.2.1 Game Theoretic Approach

We use the synthesis tool by Piterman et al. [71] which solves the synthesis problem by viewing it as the solution of a two-player game. The synthesized FDS is the controller of the system and acts against its adversary, the environment. The output variables of the FDS directly control the system and thus it is viewed as part of the system. The game is then between the controller and the environment with the utility function being formulated in terms of the behavior of the system: The goal of the game for the system is to satisfy its specification, which the environment tries to prevent. For an introduction into game theory see for example the book by Nisan et al. [65].

\(^4\)The complexity classes 2EXPTIME and 3EXPTIME contain algorithms with worst-case complexity \(2^{2p(n)}\) and \(2^{2^{2p(n)}}\) respectively, where \(p(n)\) is a polynomial in the input size \(n\).
Consider the FDS $\mathcal{M}$ in Figure 2.6. It represents the synthesized controller with inputs $\mathcal{X}$ (the environment variables) and outputs $\mathcal{Y}$ (the system variables). The environment and the controller are allowed to influence exclusively the environment and system variables respectively. From the point of view of the controller, these can be seen as the measured state of the system and the inputs to the system respectively. Thus the controller can “react” to its observed changes in the environment variables by changing the system variables.

The controller and the environment are taking turns in the game. Due to the adversarial nature of the environment it is assumed to always take the worst possible move evaluated against the utility function of the controller. The environment is also thought of as having perfect knowledge about the controller’s state and strategy, i.e. its transition relation. However, the controller only knows the current observable state of the environment, and possibly the history of environment states that the controller has stored.

The utility function of the controller can only take one of two values, corresponding to satisfying and violating its specification. The synthesis procedure therefore tries to find a winning strategy for the controller, implemented as an FDS, that maximises this utility by ensuring that the specification is never violated. Such a winning strategy only exists if the specification is realizable. The possible solutions are equilibria in which both the controller and the environment cannot improve their outcome by changing their respective strategy (i.e. Nash equilibria). However, these strategies are by no means unique, and thus several FDS’s implement the same specification.

**Closed and Open Systems**

A system that does not interact with its environment is called **closed**. The computations of such a system do not depend on the environment, and therefore this is modelled by having an empty set of environment variables $\mathcal{X}$. A system that generates its inputs nondeterministically by itself can also be regarded as a closed system.

In contrast, a system interacting with its environment is called **open**. This is modelled by having a nonempty set of environment variables $\mathcal{X}$. For the system to be open it is necessary that it does not control its environment, in the sense that it cannot directly influence its environment variables.

For the synthesis of an open system, it is irrelevant how the variables in $\mathcal{X}$ are controlled – they might be controlled partially or entirely by another FDS as shown by $\mathcal{P}$ in Figure 2.7. However, all the variables that are not controlled
by the system are considered to be part of one single environment. One way to close an open system, which is necessary for simulation (see Section 5.1.1), is by synthesizing an FDS (or a set of FDS’s) obeying the environment assumptions and connecting it to the system. This is explained in detail in Section 2.3

Figure 2.7: The controller \( M \) with its environment, \( P \). Note that \( P \) does not need to be an FDS.

### 2.2.2 Computational Considerations

Synthesis of LTL formulae for general specifications is 2EXPTIME-complete [71]. However, for a subset of assume-guarantee specifications called Generalized Reactivity[1] (GR[1]), synthesis can be performed in cubic time in the number of conjuncts in the specification and in the number of possible valuations of the system and environment variables (Theorem 1, [71]). GR[1] formulae are explained in detail in Section 3.1.4.

While the implementation of an FDS would typically be run on an embedded processor with lower computational power, a high-performance computer can be used for synthesis. Thus, efficiency of executing an FDS is often more important than how much time and memory it takes to synthesize. Once an FDS is loaded into memory, transitions can be performed in \( O(1) \) time.

One of the main considerations is therefore the space requirements of the synthesized FDS. The larger the state space, the more memory is required for storage. In the worst case, the space requirements grow quadratically with the size of the state space. This is the case when the graph representing the FDS’s transition relation is dense, i.e. when there are transitions between almost any two states. Sometimes this can be alleviated by strengthening the assumptions on the environment, since then the transitions can be removed that are precluded by the assumptions, and would never be taken under correct execution.

Note that if an environment variable changes so that no transition is available, the FDS blocks. This can only be the case if the environment assumptions are violated, and so the specification still holds even in that case. While this is important to be able to reduce the amount of available transitions during synthesis, such behavior of course is not useful in practice.
2.3. COMPOSITIONAL SYNTHESIS

2.2.3 Synthesis Software

The synthesis tool used in this thesis is based on the work by Piterman et al. [71]. Synthesis of an LTL specification proceeds in several steps.

First, a Game Structure is constructed from the specification, separating the initial conditions, the formulae specifying the transition relation, and the formulae for the game’s goals. Then, the realizability of the specification is checked by solving a $\mu$-calculus formula [45] that corresponds to the Game Structure [38]. This can be done by the Temporal Verification System (TLV), a software tool for computer-aided verification [78]. Once the $\mu$-calculus formula has been successfully solved (which can only be done if the specification is realizable), the intermediate values from the solution process can be used to construct an FDS satisfying the specification, which is again done by invoking TLV. Piterman et al. describe the details of these back-end activities [71].

The front-end that we use is TuLiP, a Python-based software toolbox for the synthesis of embedded controllers developed by Wongpiromsarn et al. [91]. Given an LTL specification, it provides the appropriate inputs for the synthesis to TLV. Its backend is based on JTLV, a Java-based implementation of TLV [77]. TuLiP supports the synthesis of FDS’s from GR[1] specifications. This restricts the set of realizable formulae to those that are in this particular syntactic form. Also, the JTLV backend works with Binary Decision Diagrams that sometimes cannot resolve formulae that are in the form $\Box \Diamond p$.

The specifications that we develop in this thesis were encoded into the appropriate format for TuLiP to process them. TuLiP is then invoked simply by Python function calls.

2.3 Compositional Synthesis

In the previous section, the synthesis of a GR[1] formula into a single FDS has been considered. In this case, the synthesis is based on a two-player game of the system against the environment. However, it may sometimes be necessary to synthesize several reactive systems that together satisfy one global specification $\varphi$. This is called compositional synthesis. It is required when several players need to coordinate with each other in order to “beat” the environment.

Motivations for Compositional Synthesis

An FDS represents the program of a physical or logical entity. For example, an FDS might be executed as a thread on a processor. If there are several FDS’s that are connected with each other, they are also called components. The connections between the components are described by an architecture, which is formally defined in Section 2.3.3. When considering one component in the architecture, all other components will be called its peer components or simply its peers.
2.3. COMPOSITIONAL SYNTHESIS

Each component might be executed on a different processor, or as a different thread on the same processor. This imposes restrictions on which variables can be accessed and modified by each component. Often the physical topology naturally imposes restrictions on the architecture supplied to the synthesis procedure.

The specification $\varphi$ provided to the synthesis procedure is an LTL formula over variables in some set $V$ which is partitioned into $X$ and $Y$, the global environment variables and the global system variables respectively. Synthesizing a single FDS means that all system variables in $Y$ can be freely manipulated without any constraints. If there are several components, then the components may only manipulate disjoint subsets of $Y$. This avoids inconsistencies by allowing each variable in $Y$ to be manipulated by at most one component. However, each variable in $V$ may be read by several components.

A specification $\varphi$ need not refer to all variables in $Y$. It might be necessary to introduce additional variables that are only relevant locally to the individual components (cf. Section 2.3.5) in order to implement communication protocols between the components. Such protocols are of no concern in the global specification and hence are abstracted away. Still, it is necessary that these additional variables are treated as system variables, and thus are in the set $Y$.

2.3.1 Game Theoretical Approach

Compositional synthesis can be seen as solving a multi-player game by finding a set of strategies that obey both the game’s rules (the global specification) and the topology of communication (the architecture). The players should cooperate with each other in order to beat an adverse environment.

Synthesizing such cooperative strategies is hard and in general undecidable [76]. Another way to approach this problem is to recast it into a set of two-player games in which each component acts against all other components and the environment. Since cooperation can be achieved by negotiating according to predefined protocols, the set of realizable global specifications is limited.
Negotiation is part of a strategy and thus can be treated using the same kind of specifications as in the rest of the strategy. It is required to establish cooperation patterns such as deciding which robots should rescue a target and which ones should keep searching. When synthesizing strategies for multiple two-player games, the negotiation channels, i.e., who negotiates with whom and for what, must be explicitly encoded in each component’s specification. This limits the scope of negotiation, but will lead to a substantial reduction of the state space. This is because the components can make local decisions that together satisfy the global specification, and thus lead to the system maximizing its utility. This is ensured by decomposing the specification appropriately before synthesis. Thus the decomposition into multiple two-player games is not done by the synthesis procedure, as, in fact, this would still be undecidable.

2.3.2 Computational Considerations

The two main considerations in compositional synthesis are realizability and the size of the state space. A specification might be realizable for a single FDS, but not for a distributed architecture. This results from the restrictions on which variables can be controlled by each individual component, which may severely limit the extent of coordination between the components that can be mustered to beat the environment.

State Space Reduction

Compositional synthesis is a reasonable approach to the state space explosion problem. A single FDS with $n$ boolean variables may contain up to $O(2^n)$ states. If the system can be decomposed into $N$ subsystems of $n/N + m$ variables, where $m$ is the number of variables needed for communication in each component, then each component would contain only up to $O(N^{\sqrt{2n/2^m}})$ states.

If $m$ is small enough, this can result in fewer states per component. However, if each component receives full information about the state of each of its peers, $m$ has to be large and this might cause even more states in each component. This is because in this way each component has full global information and additional variables are required for reliable transmission, and hence the decomposition does not result in a reduction of the state space. By fixing negotiation patterns, it is actually possible to satisfy the global specification by using only local information.

Therefore, it might sometimes be desirable to decompose an FDS even if no physical communication restrictions are present. In fact, this might be the only way to practically synthesize a specification within reasonable time and for a given amount of memory.
2.3. COMPOSITIONAL SYNTHESIS

Realizability
When reasoning about the realizability of a specification in compositional synthesis, the required architecture has to be taken into account. The architecture imposes constraints on the variables that each component can manipulate and thus not all global system variables can be arbitrarily influenced, as is the case in a centralized design.

Realizability of a specification depends primarily on the architecture and how the components communicate. If the architecture is in form of a “pipe”, i.e. a purely serial interconnection in which only the first component receives inputs from the environment, then realizability can be decided. However, even for architectures not much more complicated, for example those involving feedback, realizability is undecidable [56, 76].

That does not mean that no feedback communication is realizable, but that no algorithm can be expected to decide realizability for any given specification and architecture. By performing the decomposition into specifications satisfying the architecture before synthesis, many global specifications can be realized, even if distributed synthesis of the global specification is undecidable.

2.3.3 Architectures

Compositional synthesis requires not only a global specification $\varphi$, but also information on how to split up the tasks between the players and their means of communication. This is captured by an architecture $\mathcal{A}$. The synthesizer therefore gets both the specification $\varphi$ and the architecture $\mathcal{A}$, and outputs a whole set of FDS’s $\mathcal{M} = \{\mathcal{M}_1, \mathcal{M}_2, \ldots\}$ that together satisfy the specification and the architecture. This is illustrated in Figure 2.8.

Definition 5. An architecture $\mathcal{A}$ is a set of pairs $\{(\mathcal{X}_i, \mathcal{Y}_i) \mid i \in I\}$, where $I \subseteq \mathbb{N}$ is a finite set, indexing the components, for which the following restrictions hold:

(A1) \hspace{1em} $\forall i \in I. (\mathcal{X}_i \subseteq V) \land (\mathcal{Y}_i \subseteq V)$, all variables in the architecture are global.

(A2) \hspace{1em} $\forall i, j \in I. (i \neq j) \Rightarrow (\mathcal{Y}_i \cap \mathcal{Y}_j = \emptyset)$, no variable is controlled by more than one FDS.

(A3) \hspace{1em} $\forall i \in I. (\mathcal{X}_i \cap \mathcal{Y}_i = \emptyset)$, no component controls its environment.

The sets $\mathcal{X}_i$ and $\mathcal{Y}_i$ represent the environment and system variables respectively of each FDS in $\mathcal{M}$. Still, each component may have some of its environment variables controlled by the global environment, and some by peer components. The variables controlled by the global environment will be denoted $\mathcal{E}_i \subseteq \mathcal{X}_i$. The environment variables of a component $\mathcal{M}_i$ controlled by its peer $\mathcal{M}_j$ will be denoted by $\mathcal{T}_{i,j}$ and are called transmission variables $\footnote{Clearly $\mathcal{T}_{i,i} = \emptyset$ for all $i$, because no component controls its own environment.}$. Given an architecture, the state variables are $V = \bigcup_{i \in I} (\mathcal{X}_i \cup \mathcal{Y}_i)$, the global environment variables...
2.3. COMPOSITIONAL SYNTHESIS

![Diagram of two FDSs being composed](image)

**Figure 2.9:** Composition of two components into a composite FDS.

can be obtained from \( \mathcal{X} = \bigcup_{i \in I} \mathcal{X}_i \setminus \bigcup_{j \in I} \mathcal{Y}_j \), and the global system variables are \( \mathcal{Y} = V \setminus \mathcal{X} \). Also, \( \mathcal{E}_i = \mathcal{X}_i \setminus \bigcup_{j} \mathcal{Y}_j \) and \( \mathcal{T}_{i,j} = \mathcal{X}_i \cap \mathcal{Y}_j \). The variables in \( \mathcal{S}_i \) are controlled by \( \mathcal{M}_i \) and represent the system variables that are relevant in the global specification.

The set of components \( \mathcal{M} = \{ \mathcal{M}_i | i \in I \} \) is said to satisfy a given architecture \( \mathcal{A} = \{ (\mathcal{X}_i, \mathcal{Y}_i) | i \in I \} \) if the input variables of \( \mathcal{M}_i \) are \( \mathcal{X}_i \), its output variables are \( \mathcal{Y}_i \) and its state variables are \( \mathcal{X}_i \cup \mathcal{Y}_i \) for all \( i \in I \). If \( \mathcal{M} \) satisfies \( \mathcal{A} \), then this is written as \( \mathcal{M} \models \mathcal{A} \).

Consider for example the distributed architecture in Figure 2.9. It shows two components \( \mathcal{M}_1 \) and \( \mathcal{M}_2 \) and how they are connected by the choice of variables: \( \mathcal{M}_1 \) can control the variables in \( \mathcal{Y}_1 = \mathcal{S}_1 \cup \mathcal{T}_{1,2} \), where \( \mathcal{S}_1 \) and \( \mathcal{T}_{1,2} \) may have a nonempty intersection. The variables in \( \mathcal{X}_1 = \mathcal{E}_1 \cup \mathcal{T}_{1,2} \) cannot be controlled by \( \mathcal{M}_1 \). Instead, \( \mathcal{M}_1 \) considers them to be controlled by its environment. \( \mathcal{E}_1 \) and \( \mathcal{T}_{1,2} \) are disjoint, where \( \mathcal{T}_{1,2} \) is controlled by \( \mathcal{M}_2 \). However, from the point of view of \( \mathcal{M}_1 \), it is irrelevant how the variables in \( \mathcal{X}_1 \) are controlled. Thus,

\[
\mathcal{M}_1 = (\mathcal{E}_1 \cup \mathcal{S}_1 \cup \mathcal{T}_{1,2} \cup \mathcal{T}_{2,1}, \mathcal{E}_1 \cup \mathcal{T}_{1,2}, \mathcal{S}_1 \cup \mathcal{T}_{2,1}, Q_1, \Theta_1, \rho_1) \tag{2.2}
\]

and symmetrically

\[
\mathcal{M}_2 = (\mathcal{E}_2 \cup \mathcal{S}_2 \cup \mathcal{T}_{1,2} \cup \mathcal{T}_{2,1}, \mathcal{E}_2 \cup \mathcal{T}_{1,2}, \mathcal{S}_2 \cup \mathcal{T}_{2,1}, Q_2, \Theta_2, \rho_2). \tag{2.3}
\]

2.3.4 Composition

An architecture merely describes the connections between the individual components. As stated above, compositional synthesis has to produce a set of components that satisfy both the architecture and the global specification \( \varphi \). So far, we have only defined what it means for a single FDS to satisfy its specification. Therefore a formal description of the behavior of a set of components is required. The idea is to compose the FDS’s in \( \mathcal{M} \) into one larger FDS \( \mathcal{M} \) that generates computations over the variables \( V \). This FDS is called the composite FDS.

In a **synchronous composition**, the FDS’s proceed in lock-step, i.e., they all make a transition at the same time. This would be reflected by the composite FDS \( \mathcal{M} \) making a transition if and only if all FDS’s in \( \mathcal{M} \) make a transition. Considering that typically each FDS is executed on an individual processor,
2.3. COMPOSITIONAL SYNTHESIS

this requires their clocks to be precisely synchronized and communication to be implemented with negligible latency. These assumptions can usually not be satisfied in practice, but still this form of composition is often a topic of theoretical interest [20, 73].

Instead, in an asynchronous composition the execution does not have to be in lock-step and therefore no synchronization is required. Communication is still assumed to be with negligible latency, but this can often be relaxed when adapting the local specifications accordingly. The composite FDS $\mathcal{M}$ then proceeds even if only a single FDS $M_i$ in $\mathcal{M}$ makes a transition. In order to simplify notation, the following definition of an asynchronous composition is only stated for two FDS’s, but it can easily be extended for an arbitrary number of FDS’s.

**Definition 6.** The asynchronous composition of the FDS’s

$$
\mathcal{M}_1 = (\mathcal{E}_1 \cup S_1 \cup T_{1,2} \cup T_{2,1}, \mathcal{E}_1 \cup T_{1,2}, S_1 \cup T_{2,1}, Q_1, \Theta_1, \rho_1),
$$

$$
\mathcal{M}_2 = (\mathcal{E}_2 \cup S_2 \cup T_{2,1} \cup T_{1,2}, \mathcal{E}_2 \cup T_{2,1}, S_2 \cup T_{1,2}, Q_2, \Theta_2, \rho_2)
$$

is the composite FDS

$$
\mathcal{M}_1 | \mathcal{M}_2 = (V_1 \cup V_2, \mathcal{E}_1 \cup \mathcal{E}_2, Y_1 \cup Y_2, Q_1 \times Q_2, \Theta_1 \wedge \Theta_2, \rho),
$$

where the transition relation is a function $\rho : Q_1 \times Q_2 \rightarrow 2^{Q_1 \times Q_2}$ such that for all $(q_1, q_2), (q'_1, q'_2) \in Q_1 \times Q_2$,

$$
(q'_1, q'_2) \in \rho(q_1, q_2) \iff q'_1 \in \rho_1(q_1) \wedge (q'_2 = q_2) \vee
q'_2 \in \rho_2(q_2) \wedge (q'_1 = q_1) \vee
q'_1 \in \rho_1(q_1) \wedge q'_2 \in \rho_2(q_2),
$$

or equivalently

$$
(q_1, q_2) \rightarrow_{\rho} (q'_1, q'_2) \iff (q_1 \rightarrow_{\rho_1} q'_1) \wedge \mathcal{V}_{T_{1,2}}(q'_1) = \mathcal{V}_{T_{1,2}}(q_2) \wedge (q_2 = q'_2) \vee
(q_1 = q'_1) \wedge (q_2 \rightarrow_{\rho_2} q_2) \vee
(q_1 \rightarrow_{\rho_1} q'_1) \wedge \mathcal{V}_{T_{2,1}}(q'_2) = \mathcal{V}_{T_{2,1}}(q_1) \wedge (q_2 \rightarrow_{\rho_2} q'_2).
$$

Note that in order to model asynchrony, the values of the transmission variables $T_{1,2}$ and $T_{2,1}$ are only equated between $q_1$ and $q_2$ in a transition of $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively.

The asynchronous composition is in this thesis also referred to simply as composition. Composition is both associative and commutative. That is, $\mathcal{M}_1 | \mathcal{M}_2 = \mathcal{M}_2 | \mathcal{M}_1$ and $(\mathcal{M}_1 | \mathcal{M}_2) | \mathcal{M}_3 = \mathcal{M}_1 | (\mathcal{M}_2 | \mathcal{M}_3)$ for any components $\mathcal{M}_1, \mathcal{M}_2$ and $\mathcal{M}_3$. 

\[6\] The standard definition of asynchronous composition prohibits two FDS’s to make a synchronous transition.
and \( M_3 \) with appropriate environment and system variables. Thus, the parentheses are usually dropped and an arbitrary (finite) number of components can be composed to obtain a composite FDS.

Defining the composition of FDS’s to be an FDS in its own right has one major advantage: Since LTL specifications are statements over sequences, the global specification \( \varphi \) can directly be used to make statements about the composition. For example, for the composite FDS \( M = M_1 | M_2 | \cdots | M_n \), the statement \( M \models \varphi \) is already defined to mean that \( \forall \sigma \in L_{M_1 | M_2 | \cdots | M_n} \cdot \sigma \models \varphi \).

The definition of an architecture allows some variables to be system variables of one component and environment variables of its peers. These transmission variables are used in communication protocols and the behavior of the composite FDS \( M \) depends on when a change in these variables is seen by the peer components.

### 2.3.5 Decomposition

Distributed synthesis is based on a single global specification \( \varphi \) and an architecture \( \mathcal{A} \). The specification describes how the asynchronous composition of all components in the architecture has to behave, but makes no explicit statements about the behavior of the individual FDS’s.

The compositional synthesis problem is to find a set of FDS’s so that their asynchronous composition satisfies the global specification. If the architecture defines decoupled or weakly coupled components (i.e. there are no or few transmission variables), this can be done without introducing too many additional states in the synthesized components. Since general compositional synthesis is very hard and might even be undecidable, compositional synthesis algorithms are typically built around the synthesis of a single FDS from its local specification. A separate decomposition algorithm is used to obtain the FDS’s satisfying \( \mathcal{A} \) from the global specification.

Two principal approaches for decomposition exist: In postdecomposition, the global specification is synthesized into a single FDS which is then decomposed according to the architecture (see Figure 2.10). In predecomposition, the global specification is decomposed first into several local specifications that are then synthesized separately (see Figure 2.11). Both post- and predecomposition require the resulting synthesized components to satisfy the architecture.

The approach that is adopted in this thesis is predecomposition. Let \( \varphi \) be the global specification, and \( \mathcal{A} \) be an architecture. A set of local specifications \( \{ \varphi_i \mid i \in I \} \) is a predecomposition of a global specification \( \varphi \) if any set of FDS’s \( M = \{ M_i \mid i \in I \} \) that satisfies the architecture \( \mathcal{A} \) and for which \( M_i \models \varphi_i \) for all \( i \in I \), the asynchronous composition \( M_1 | M_2 | \cdots | M_n \) satisfies the global specification \( \varphi \). That is, the local specifications \( \varphi_1, \varphi_2, \ldots, \varphi_n \) must satisfy

\[
(PD) : \forall M = \{ M_i \mid i \in I \} . M \models \mathcal{A} \land (\forall i \in I . M_i \models \varphi_i) \Rightarrow M_1 | M_2 | \cdots | M_n \models \varphi.
\]
2.3. COMPOSITIONAL SYNTHESIS

![Diagram](image)

Figure 2.10: Outline of Postdecomposition. $S$ is the synthesizer and $D_{\text{post}}$ performs the postdecomposition.

![Diagram](image)

Figure 2.11: Outline of Predecomposition. $D_{\text{pre}}$ performs the predecomposition, and then the same synthesizer $S$ is invoked several times.
2.3. COMPOSITIONAL SYNTHESIS

The decomposition of a global specification into such local specifications is discussed in Section 3. To our knowledge no general systematic approach has yet been developed. The undecidability of compositional synthesis for almost all architectures makes a fully automated approach unlikely.

This is why in the next chapter we develop approaches to manually decomposing the global specification. While we consider the restrictions imposed by the architecture on the global system variables that occur in the global specification, the architecture may be enhanced with additional transmission variables if this is needed.
Chapter 3

Specification

The reactive controllers of the SAR robots decide on the actions of the robots in order to find and rescue the targets. As described in Section 2.2.1 and Section 2.3.1, the controllers are thought of as players in a game that is subject to a set of rules, has a goal and an initial configuration, similar to a board game. However, board games are usually described informally in prose, which is prone to ambiguities, contradictions and omissions.

Instead, formal specifications can be used to precisely describe the behavior of software and hardware systems \[11, 33, 82\]. The two most common scenarios are to use the specification to verify a given system (see Model Checking in Section 2.1.3) or to synthesize a system that satisfies a specification by construction (see Section 2.2).

A specification language should facilitate the development of specifications satisfying the following requirements \[74\]:

**Incremental Modification** Once a specification is finalized, it should be easy to add another requirement, or relax an existing one.

**Consistency** There should exist at least one system that can actually satisfy the implementation. This property does not, however, state that such a system can be found in a straightforward manner.

**Completeness** A specification should include all important properties the system has to satisfy.

**Validation** From a specification, it should be possible to check whether an implementation satisfies the designer’s intentions. This is not the same as verification, since an implementation that satisfies a formal specification might still not do what the designer wants. Validation is an informal process, and so the specification language should be semantically closely related to the specification-relevant subset of natural language.

**Scalability** The specification language should be suitable for large systems.

Incremental modification, consistency and validation are all viable in LTL. However, the completeness of an LTL specification is very difficult to verify, which
also restricts the scalability of specifications. Notwithstanding these limitations, LTL has become a common way to express properties of reactive systems. We therefore investigate how to use LTL as a specification language for controllers that fit into the game-theoretic paradigm, as well as taking into account the input-to-output characteristics of standard controllers.

A specification for an FDS $M = (V, X, Y, Q, \Theta, \rho)$ is an LTL formula $\varphi \in \mathcal{L}(V)$ over the variables $V$. Similarly, a program then satisfies the specification if $\varphi$ is valid for any FDS $M$ describing the program. Note that there might be several FDS’s describing the same program and vice versa.

A property over the variables $V$ is a subset of the sequences in $\mathcal{V}(V)$. An LTL formula $\varphi \in \mathcal{L}(V)$ then expresses the property $\{ \sigma \in \mathcal{V}(V) \mid \sigma \models \varphi \}$. Not every property can be expressed by an LTL formula, since the properties expressible by LTL are countable, but the set of all properties, $2^\mathcal{V}(V)$ is not. Also, the expressivity of LTL is limited by the (bounded) nesting depth of the next operator.

Since we are only concerned with properties expressible in LTL, we use the terms property and formula fairly interchangeably if the context is clear. Also, given a formula $\varphi$, the corresponding property can be found by forming the set $\{ \sigma \in \mathcal{V}(V) \mid \sigma \models \varphi \}$.

In this and the following chapters, intervals of integers are denoted in the following way: $[a, b] = \{ x \in \mathbb{Z} \mid a \leq x \leq b \}$ and $(a, b) = \{ x \in \mathbb{Z} \mid a < x < b \}$ for integers $a$ and $b$ such that $a < b$. Also, the boolean domain $\{0, 1\}$ is denoted by $\mathcal{B}$.

Often, integer variables are used to represent data storage. To represent that a storage is empty, the integer $-1$ may be used. For better readability we use $\epsilon$ as a synonym for $-1$, and so the intervals $[\epsilon, n)$ and $[\epsilon, n]$ are the same as $[-1, n)$ and $[-1, n]$.

### 3.1 Classification of Formulae

Specifications for reactive systems typically consist of both safety and liveness formulae. This classification was first introduced by Lamport in 1977. Later, Manna and Pnueli gave an entire taxonomy in the form of a “hierarchy” to classify different formulae.

This classification also contains elaborate formulae such as assumption/guarantee (A/G) and GR[1] specifications, which are introduced in Section 3.1.3 and Section 3.1.4 respectively. One important attribute “orthogonal” to the classification between safety and liveness is whether the sequences satisfying a formula are invariant under finite repetitions of elements, which is discussed in Section 3.1.5.

---

1 Here $2^S$ denotes the powerset of a set $S$. 

32
3.1. CLASSIFICATION OF FORMULAE

3.1.1 Safety and Liveness Properties

Informally, a **safety property** states that “nothing bad will ever happen.” In the specification of control systems, safety properties ensure that constraints are never violated. A formula \( \varphi_S \) expressing a safety property then states that all finite prefixes of a computation \( \sigma \) satisfy some propositional formula \( p \).

According to Sistla [83], \( \varphi_S \) is a **safety formula** iff all sequences \( \sigma \) for which all finite prefixes can be extended to an infinite sequence satisfying \( \varphi_S \), satisfy \( \varphi_S \). Denoting the set of finite prefixes of a sequence \( \sigma \) by \( \text{pref}(\sigma) \), this statement can be written as

\[
(\forall \sigma \in \mathcal{V}(V)\}. (\forall \vec{\sigma} \in \text{pref}(\sigma)). (\exists \sigma \in \mathcal{V}(V)\}. (\vec{\sigma} \sigma \models \varphi_S) \Rightarrow \sigma \models \varphi_S.
\]

Any safety formula can be written in the form \( \Box p \), where \( p \) is a past formula.

Deciding whether a given set of sequences is a safety property is PSPACE-complete ([83], Theorem 5.5). However, it has been shown that safety formulae are closed under \( \land, \lor, \bigcirc, \mathbb{U} \) and \( \Box \) ([83], Theorem 3.1). Of course, then safety is also preserved under weak until \( \mathbb{U} \), but not necessarily under strong next \( \bigcirc \) for finite sequences. These syntactic properties provide an efficient way to decide whether a given formula expresses a safety property.

A **liveness property** states that “something good will eventually happen.” Therefore, a **liveness formula** \( \varphi_L \) expressing a liveness property requires that every finite sequence \( \sigma \) can be extended to satisfy \( \varphi_L \). That is, if \( \sigma \not\models \varphi_L \) then there exists a possibly infinite sequence \( \vec{\sigma} \) so that the concatenation of the two sequences \( \sigma\vec{\sigma} \) satisfies \( \varphi_L \). Formally, \( \varphi_L \) is a liveness formula iff

\[
(\forall \sigma \in \mathcal{V}(V)^+. (\sigma \not\models \varphi_L \Rightarrow \exists \vec{\sigma} \in \mathcal{V}(V)^+. (\sigma\vec{\sigma} \models \varphi_L)).
\]

Liveness formulae are necessary in the specification of reactive control systems in order to ensure that the controller maintains an ongoing interaction with its environment. If no liveness formula is included in a specification then it can be satisfied by sequences consisting only of the initial state (or a finite number of states.) This becomes an issue when synthesizing an FDS from such a specification not containing a liveness formula, since the synthesis algorithm will try to generate the simplest possible FDS, which does not need to be nonblocking in this case.

3.1.2 Derived Classes

Several important and widely-used classes can be derived from the safety and liveness classification [57, 90]. We present these classes using past formulae \( p \) and \( q \). However, since we only consider future formulae for synthesis, \( p \) and \( q \) are propositional formulae (which are both past and future formulae).

A **guarantee** formula is of the form \( \Diamond p \). It is therefore a liveness formula and expresses that the system must eventually satisfy the formula \( p \), e.g. termination in a total correctness specification of a program.
3.1. CLASSIFICATION OF FORMULAE

An obligation formula is a combination of safety and liveness formulae of the form $\Box p \lor \Diamond q$. Such formulae can specify the behavior in exceptional circumstances. For example, $\Box \neg p \lor \Diamond (p \leftrightarrow q)$ ensures that either the exception $p$ never happens and if it does, it is handled correctly by $q$.

The class of obligation formulae strictly contains the safety and liveness formulae and so a general obligation formula cannot be specified by either a safety or a liveness formula. Every boolean combination of safety and guarantee formulae is an obligation formula.

A progress or recurrence formula is of the form $\Box \Diamond p$ and expresses that $p$ has to be satisfied at infinitely many positions. If $p$ is a propositional formula, then a progress formula is a liveness formula. However, if $p$ is allowed to be any past formula, then the class of progress properties is strictly greater than that of obligation, safety or liveness properties.

A response formula is of the form $\Box (p \rightarrow \Diamond q)$ and ensures that the formula $q$ will hold at some point in the future whenever $p$ is satisfied. Response formulae are progress formulae, and if both $p$ and $q$ are propositional formulae then they are even liveness formulae, since they ensure that $q$ is always asserted as a reaction to $p$.

A stability or persistence formula ensures that once a proposition $p$ holds, it will never be violated in the future. Such a formula is of the form $\Diamond \Box p$. It can be understood as requiring the system to enter into a stable state characterized by $p$, and remaining there. The class of stability properties is strictly larger than the class of obligation, recurrence and persistence properties.

A reactivity formula is of the form $\Box \Diamond p \lor \Diamond \Box q$ and is a combination of a progress and a stability formula. The class of reactivity properties is strictly larger than any of the abovementioned classes. An even larger class of properties is that of the general reactivity properties, which contains sequences satisfying finite conjunctions of reactivity formulae. Thus a general reactivity formula is of the form $\bigwedge_{i=1}^{n} (\Box \Diamond p_i \lor \Diamond \Box q_i)$ where $p_i$ and $q_i$ are past formulae. Every formula specifiable in LTL can be expressed as a general reactivity formula [57].

The format of general reactivity formulae gives the idea for the classification of GR[1] specifications introduced below in Section 3.1.4. Since every LTL formula is expressible as a general reactivity formula, also GR[1] formulae are surprisingly expressive, even if the restrictions imposed on $p_i$ and $q_i$ are quite severe.

3.1.3 Assumption/Guarantee Specifications

When synthesizing open controllers from LTL specifications, a two-player game is considered, in which the controller plays against an adversarial environment. The controller tries to win by satisfying its specification while it assumes that the environment always tries to falsify the specification with the moves it is allowed to make.
Since the environment is considered to be adverse, it may always do the worst possible move from the controller’s point of view, open controllers cannot be expected to satisfy their specification in all environments. An open controller interacts with an environment it cannot influence, so the assumptions it has about the environment do not actually constrain the environment. The controller cannot prevent the environment from violating the assumptions but in this case it is allowed to also violate its goals.

The synthesis procedure only supports the synthesis of an FDS satisfying a single specification \( \varphi \), without any explicit assumptions on the environment. However, since the environment and system variables play different roles in the synthesis algorithm, the specification \( \varphi \) can be written in a format that makes explicit what is assumed of the environment and what is guaranteed by the controller.

Let \( \varphi^e \) be the **environment assumptions**, i.e. the formula describing exactly what the system may assume about the environment. Let \( \varphi^s \) be the **system goals**, i.e. the specification that the controller is supposed to satisfy given that the environment satisfies its assumptions. It is then sufficient for the synthesized FDS to satisfy the specification \( \varphi^e \rightarrow \varphi^s \).

Such a specification is called an **assumption/guarantee specification** (A/G specification for brevity,) and was first introduced by Misra and Chandy in 1981 [60]. The controller is implemented to satisfy \( \varphi^s \) given \( \varphi^e \), but has no knowledge about the actual implementation of the environment. It cannot assume that an actual FDS is controlling \( X \) according to \( \varphi^e \). If there was such an FDS, \( \varphi^e \) would never be violated, so it would not be necessary to even include it in the original specification. In contrast, the system only needs to satisfy \( \varphi^s \) as long as the environment satisfies \( \varphi^e \) and if \( \varphi^e \) is violated, the system is allowed to exhibit any behavior.

A/G specifications are mainly used to specify open controllers that communicate with each other. The environment assumptions of components that communicate with each other can be thought of as specifying a communication protocol. A component assumes that its peers comply with the protocol, and only if this is the case it is required to also satisfy the protocol itself. If the environment violates the protocol, the component also is not required to be able to satisfy it.

### 3.1.4 GR[1] Specifications

A/G specifications are very useful in defining open reactive systems. However, since we are concerned with synthesizing FDS’s from their specifications, the class of formulae that we can use is restricted by the capabilities of the synthesizer. Thus, even a realizable specification (i.e. one for which an FDS exists) might not be synthesizable.

Synthesis of general reactivity formulae is 2EXPTIME-complete [76], but there exist smaller classes of formulae that can still express a large class of useful properties but exhibit better computational complexity for synthesis. The class
3.1. CLASSIFICATION OF FORMULAE

of A/G formulae supported by the JTLV synthesizer is called Generalized Reactivity [1] (GR[1]). Synthesis and checking realizability of such formulae can be done in polynomial time, see Section 2.2.2. A GR[1] formula is of the form

$$\varphi^e_{\text{init}} \land \varphi^i \land \varphi^g \rightarrow \varphi^s \land \varphi^r \land \varphi^g.$$  

where the superscripts e and s stand for the environment assumptions and the system goals respectively. A GR[1] formula thus states that only if some assumptions on the environment $\varphi^e = \varphi^e_{\text{init}} \land \varphi^i \land \varphi^g$ are satisfied, then it is required that the controller satisfies the specification $\varphi^s = \varphi^s_{\text{init}} \land \varphi^i \land \varphi^g$. In particular, if the environment violates $\varphi^e$, then no restrictions on the controller apply. Moreover, once a safety property is violated, it cannot be satisfied in any future state.

In the above definition, $\varphi^e_{\text{init}}$ and $\varphi^s_{\text{init}}$ are propositional formulae over $\mathcal{X} \cup \mathcal{Y}$ and characterize the initial states of the environment and the controller respectively. Synthesizing an FDS from a GR[1] specification will determine the initial condition $\Theta$, assuming that the environment initially satisfies $\varphi^e_{\text{init}}$. Since by the startup property $\text{[SP]}$ each FDS has an initial state satisfying $\Theta$, it is necessary to have $\models \Theta \rightarrow (\varphi^s_{\text{init}} \land \varphi^s_{\text{init}})$.

Further, $\varphi^e_{i}$ and $\varphi^s_{i}$ are safety formulae of the form $\bigwedge_{i \in J} \Box B_i$, where $B_i$ is a boolean combination of atomic propositions $p$ over variables in $\mathcal{X} \cup \mathcal{Y}$ and of expressions of the form $\bigvee p$, where $p$ is an atomic proposition over variables in $\mathcal{X}$ for $\varphi^e_{i}$ and over variables in $\mathcal{X} \cup \mathcal{Y}$ for $\varphi^s_{i}$. With this part of the GR[1] formula it is possible to constrain or force transitions of an FDS by using the next operator. Note however, that in the environment assumptions nothing can be said about the next values of the system variables $\mathcal{Y}$.

Lastly, $\varphi^g_{i}$ and $\varphi^r_{i}$ are liveness formulae of the form $\bigwedge_{i \in J} \Diamond \Box B_i$ where the $B_i$ are propositional formulae over variables in $\mathcal{X} \cup \mathcal{Y}$. These properties can be used to ensure that the controller is not idle as long as the environment is making transitions.

While GR[1] formulae might seem restrictive at first sight, they allow the expression of many useful properties of reactive systems. Let

$$\phi^s = \bigwedge_{j \in I_1} \Box p_{1,j} \land \bigwedge_{j \in I_2} \Diamond p_{2,j} \land \bigwedge_{j \in I_3} (\Box p_{3,j} \lor \Diamond q_{3,j})$$

$$\land \bigwedge_{j \in I_4} \Box \Diamond p_{4,j} \land \bigwedge_{j \in I_5} (\Box p_{5,j} \rightarrow \Diamond q_{5,j}) \land \bigwedge_{j \in I_6} \Diamond \Box p_{6,j}$$

be a formula composed of safety, guarantee, obligation, progress, response and stability formulae, where $p_{k,j}$, $q_{3,j}$ and $q_{5,j}$ are propositional formulae over the variables in $\mathcal{V}$. Further, let

$$\phi^e = \bigwedge_{i \in I_1} \Box p_{1,i} \land \bigwedge_{i \in I_2} \Diamond p_{2,i}$$

be a formula composed of safety and progress properties, where $p_{k,i}$ are propositional formulae over the variables in $\mathcal{X}$, and let $\varphi_{\text{init}}$ be a propositional formula over $\mathcal{V}$. It has been shown that any formula of the form $(\varphi_{\text{init}} \land \phi^e) \rightarrow \phi^s$ can be
3.1. CLASSIFICATION OF FORMULAE

translated into a GR[1] formula (50, Proposition 1). For example, a response
property \(\Box(p \rightarrow \Diamond q)\) that has to be satisfied by the system can be translated
into a GR[1] formula by introducing an auxiliary boolean variable \(t\) and letting
\[
\begin{align*}
\varphi^s_{\text{init}} &= \neg t \rightarrow (p \land \neg q) \\
\varphi^s_t &= \Box((\neg t \lor \Diamond p) \land \Diamond \neg q) \rightarrow \Diamond \neg t) \\
&\quad \land \Box(\neg t \land \Box \Diamond t \rightarrow \Diamond q) \\
&\quad \land \Box(t \land \Box \neg t \rightarrow \Diamond p) \\
\varphi^s_g &= \Box \Diamond t.
\end{align*}
\]
The variable \(t\) is called a **trigger**, because it triggers \(q\) to be asserted eventu-
ally. The trigger is asserted as long as there is no obligation to assert \(q\). Thus,
when \(t\) is lowered, the system must assert \(q\). Initially, “\(t = \text{True}\)” holds un-
less \(p \land \neg q\), the only case in which there is an obligation to eventually raise
\(q\). The three conjuncts of \(\varphi^s_t\) ensure that \(t\) is only changed appropriately.
\(\Box((\neg t \lor \Diamond p) \land \Diamond \neg q) \rightarrow \Diamond \neg t)\) ensures that \(t\) is lowered when \(p\) is asserted
but \(q\) is not, and that \(t\) stays low as long as “\(q = \text{False}\)” holds. The formulae
\(\Box(\neg t \land \Box t \rightarrow \Diamond q)\) and \(\Box(t \land \Box \neg t \rightarrow \Diamond p)\) ensure that the trigger is only raised
when \(q\) is high and lowered only when \(p\) is high. Finally, the progress formula
\(\varphi^s_g\) ensures that the system will always eventually reach a state where it is no
longer under an obligation to raise \(q\).

3.1.5 Stutter-Invariance

Stuttering is a phenomenon connected with the repetition of states in the com-
putations of an FDS. Two sequences \(a = a_0a_1a_2\ldots \) and \(b = b_0b_1b_2\ldots\) are said to
be **stutter-equivalent** if there exist two infinite sequences of strictly increasing
indices \(0 = i_0 < i_1 < i_2 < \cdots\) and \(0 = j_0 < j_1 < j_2 < \cdots\) such that for every
\(k \geq 0\), the sequences
\[
\begin{align*}
a_{i_k}a_{i_k+1}\ldots a_{i_k+1-1} \\
\text{and} \quad b_{j_k}b_{j_k+1}\ldots b_{j_k+1-1}
\end{align*}
\]
are identical [69]. An LTL formula \(\varphi\) is **stutter-invariant** if for any two stutter-
equivalent sequences \(a\) and \(b\), either both \(a \models \varphi\) and \(b \models \varphi\) or both \(a \not\models \varphi\) or \(b \not\models \varphi\).2

Informally, if a sequence satisfies a stutter-invariant formula, the same sequence
with any element duplicated any number of times also satisfies the same formula.

Stutter-invariance can be an important feature of specifications of an FDS. In a
clock-triggered FDS, at each clock tick the values of the environment variables
are read, and the next state is selected so that the environment variables have
exactly these values. However, it might be the case that the values of the en-
vironment variables have not changed from one clock tick to the next. This is
called environment stuttering.

In an asynchronous composition of several components, it is possible that one
component proceeds more than once while all its peers as well as its environ-
ment stay in the same state. The inputs are observed to be repeated. When the

\[2\text{That is, } \varphi \text{ is satisfied by the union of stutter-equivalence classes.}\]
3.1. CLASSIFICATION OF FORMULAE

components are executed on different processors or as different threads on the same processor, this situation occurs naturally from thread scheduling, different clock speeds and clock rate drift [44].

A clock-triggered FDS must therefore be invariant to environment stuttering, requiring the environment assumptions to be stutter-invariant. However, even if the system goals are stutter-invariant, the synthesis algorithm always tries to minimize system stuttering, since this would increase the state space.

Peled and Wilke have shown that stutter-invariant properties in (future) LTL are exactly those that can be expressed without the next operator (i.e. the only allowed temporal operator is the until operator) and that stutter-invariance is decidable ([69], Lemma 2 and Corollary 6). Moreover, PSPACE methods exist for determining stutter invariance [70].

To obtain stutter-invariant environment assumptions, it is therefore sufficient to avoid the next operator. However, the synthesis method used in this thesis only supports the temporal operators always, eventually and next. Since the until operator is not supported, it is sometimes necessary to express formulae using the next operator. However, careful construction of these formulae might still render them expressible without the next operator.

For example, consider the formula

\[ \Box (x \rightarrow \Diamond (\neg x \rightarrow y)) \]

expressing that \( x \) may only be cleared if \( y \) is asserted in the next state. This is a synthesizable property as it can be included in either \( \varphi_e^t \) or \( \varphi_s^t \) in a GR[1] specification. Moreover, it can be expressed without using the next operator by

\[ \Box (x \rightarrow x U y) \Leftrightarrow \Box (x \rightarrow (x U y \lor \Box x)), \]

and is therefore stutter-invariant.

3.1.6 Specification Example

In order to illustrate the specification concepts above, a toy example is presented. A GR[1] specification is given that is synthesized by TuLiP into an FDS.

The controller has one boolean environment variable \( x \) and two boolean system variables \( y_1 \) and \( y_2 \). First the guarantee part of the specification is given. The system should start with both \( y_1 \) and \( y_2 \) cleared and then respond to an input \( x \) by raising \( y_1 \):

\[ \Box (x \rightarrow \Diamond y_1). \]

This can be translated into a GR[1] formula by the method shown in Section 3.1.4.

Note that this will introduce another boolean system variable, called a trigger

\(^{3}\text{Note that } \Box \varphi \text{ is equivalent to } \neg (\text{True} U \neg \varphi) \text{ and therefore expressible without using } \Box. \]

38
that we denote by \( t \) in this example. We demand that \( y_1 \) may only be raised if \( y_2 \) is high. This is expressed by the guarantee

\[
\square (\neg y_1 \land \bigcirc y_1 \rightarrow y_2).
\]

To prevent \( y_2 \) from being raised arbitrarily, \( x \) is required to be cleared as a prerequisite:

\[
\square (\neg y_2 \land \bigcirc y_2 \rightarrow \neg x).
\]

This specification consisting of just these three guarantees is unrealizable. This is because \( x \) might be asserted initially, requiring \( y_1 \) to be raised eventually. But if the environment never clears \( x \), then \( y_2 \) cannot be raised, which is required for \( y_1 \) to fulfill the response formula.

In order to make the specification realizable, an assumption on the environment has to be included. Assuming that \( x \) is always cleared eventually turns out to be sufficiently strong. It is expressed by the progress formula

\[
\square \Diamond \neg x.
\]

The symmetric assumption that \( x \) will always be asserted eventually is included as well, because the environment should never just keep \( x \) low at all times:

\[
\square \Diamond x.
\]

Thus, the complete GR[1] specification is

\[
\varphi = (\square \Diamond x \land \\
\square \Diamond \neg x) \\
\rightarrow \\
(\neg y_1 \land \neg y_2 \land \\
\square (\neg y_1 \land \bigcirc y_1 \rightarrow y_2) \land \\
\square (\neg y_2 \land \bigcirc y_2 \rightarrow \neg x) \land \\
\square (x \rightarrow \Diamond y_1))
\]

This specification is synthesizable and gives rise to an FDS \( M_\varphi \) with seven states, shown in Figure 3.1. Recall that the extra system variable \( t \) has been introduced as a trigger to express the response property as a GR[1] formula.

The initial states of \( M_\varphi \) are all those for which \( \neg y_1, \neg y_2 \) and \( \neg t \) hold (compare with the translation of the response formula into a GR[1] formula). The restrictions on the rising edges of \( y_1 \) and \( y_2 \) can be verified by tracing the corresponding arrows in the FDS. Thus \( \square (\neg y_1 \land \bigcirc y_1 \rightarrow y_2) \) is verified by noting that \( q_1 \rightarrow q_2 \) and \( q_1 \rightarrow q_3 \) are the only transitions for which \( \neg y_1 \land \bigcirc y_1 \) holds and that “\( y_2 = \text{True} \)” holds in \( q_0 \). \( \square (\neg y_1 \land \bigcirc y_1 \rightarrow y_2) \) can be similarly verified.

The response property \( \square (x \rightarrow \Diamond y_1) \) is also satisfied, but only provided that both the environment assumptions \( \square \Diamond x \) and \( \square \Diamond \neg x \) hold.\(^4\) To see this, consider all

\(^4\)The guarantees are realizable given only \( \square \Diamond \neg x \), but the synthesis procedure also makes use of \( \square \Diamond x \) to simplify \( M_\varphi \).
3.2 Specifying Distributed Systems

As explained in Section 2.3, it is often necessary to synthesize a set of reactive systems that together satisfy a given global specification. Compositional synthesis requires both a global specification and an architecture. It has already been mentioned that compositional synthesis for almost all architectures is undecidable. In this section we present principles that facilitate designing synthesizable local specifications and reasoning about the composition of the corresponding components in the architecture in order to establish that the global specification is met.

We consider predecomposition, i.e. decomposing the global specification into several local specifications that satisfy the architecture before passing each of them to the synthesizer separately. The main challenge is to perform this de-
composition in such a way that the asynchronous composition of the FDS’s synthesized from the local specifications satisfies the global specification. It is therefore necessary to have a mathematically sound way of reasoning about the asynchronous composition of the components in an architecture. Neglecting accuracy at this stage would render the efforts in synthesizing correct-by-construction controllers useless.

Several attempts have been made at providing tools for the decomposition and reasoning about the composition of specifications. Misra and Chandy were among the first to reason about the composition of A/G specifications [60]. Abadi and Lamport provide a formalism for reasoning about decomposing and composing A/G specifications in TLA [1, 2]. In response to this, a purely syntactic formalism for LTL specifications was developed by Jonsson and Tsay [36], providing a stronger composition rule. The work of Ozay et al. provides examples of decomposing GR[1] specifications and how to reason about their compositions [67, 68].

3.2.1 Composing Specifications

Before principles for decomposing specifications can be stated, it is necessary to introduce the necessary framework to reason about their composition. We present the preconditions that have to be satisfied for a set of local specifications to satisfy a global specification. For notational convenience, these conditions are stated only for two components, but the extension to more components is straightforward.

We present the Feedback Interconnection Refinement Rule stated by Ozay et al. [67]. It is a specific case of the Composition Theorem proved by Abadi and Lamport [2]. We employ this version because it uses LTL and GR[1] specifications, although it is not the most general composition rule available. The proof of Abadi and Lamport’s theorem covers several pages but the simplified version by Ozay et al. is straightforward to prove by induction on the length of the computations of the relevant FDS’s ([67], see the proof of Proposition 3). However, they do not take into account that the environment assumptions have to be stutter-invariant for the rule to hold for an asynchronous composition. The Composition Theorem by Abadi and Lamport implicitly includes this assumption, since every property expressible in TLA is stutter-invariant.

**Architecture**

Let $\mathcal{X}$ and $\mathcal{Y}$ be the global environment and system variables respectively, i.e. the variables that occur in the global specification $\varphi$. These sets are necessarily disjoint, because the environment variables $\mathcal{X}$ cannot be controlled by any component which controls $\mathcal{Y}$.

The local environment and system variables of the synthesized components $M_1$ and $M_2$ are then subsets of $\mathcal{X} \cup \mathcal{Y}$ so that even feedback between the components can be introduced (see Figure 2.4). Since all global system variables $\mathcal{Y}$
3.2. SPECIFYING DISTRIBUTED SYSTEMS

must be controlled by exactly one component, \( \mathcal{Y} \) is partitioned over the two components. Thus, the system variables of \( \mathcal{M}_1 \) are \( \mathcal{Y}_1 \subseteq \mathcal{Y} \), and the system variables of \( \mathcal{M}_2 \) are \( \mathcal{Y}_2 = \mathcal{Y} \setminus \mathcal{Y}_1 \).

The environment variables of \( \mathcal{M}_1 \) can consist of variables from the global environment, but also from the system variables of \( \mathcal{M}_2 \). Thus \( \mathcal{X}_1 \subseteq \mathcal{X} \cup \mathcal{Y}_2 \). Symmetrically, \( \mathcal{X}_2 \subseteq \mathcal{X} \cup \mathcal{Y}_1 \). In Figure 2.9, we would have \( \mathcal{E}_1 = \mathcal{X}_1 \setminus \mathcal{Y}_2 \) and \( \mathcal{T}_{2,1} = \mathcal{X}_1 \cap \mathcal{Y}_2 \) and the symmetrical relations \( \mathcal{E}_2 = \mathcal{X}_2 \setminus \mathcal{Y}_1 \) and \( \mathcal{T}_{1,2} = \mathcal{X}_2 \cap \mathcal{Y}_1 \).

This choice of variables defines the architecture for which we now develop the specification format and the hypotheses for the composition rule.

### Specification Format

Suppose that a global specification \( \varphi^e \rightarrow \varphi^s \) is a GR[1] formula, where \( \varphi^e \in \mathcal{L}(\mathcal{X}) \) and \( \varphi^s \in \mathcal{L}(\mathcal{X} \cup \mathcal{Y}) \). This is the specification that must be satisfied by the asynchronous composition of \( \mathcal{M}_1 \) and \( \mathcal{M}_2 \).

Let \( \varphi_1^e \in \mathcal{L}(\mathcal{E}_1) \) and \( \varphi_2^e \in \mathcal{L}(\mathcal{E}_2) \) be stutter-invariant LTL formulae that will be part of the environment assumptions of \( \mathcal{M}_1 \) and \( \mathcal{M}_2 \) respectively. One of the conditions that these two formulae must satisfy is that any sequence \( \sigma \) that satisfies the global environment assumptions also satisfies both \( \varphi_1^e \) and \( \varphi_2^e \). Thus

\[
(P1) \quad \forall \sigma \in \mathcal{V}(\mathcal{X})^\infty. \sigma \models \varphi^e \Rightarrow \sigma \models \varphi_1^e \land \varphi_2^e.
\]

Note that \( \land \) distributes over \( \models \), so when verifying this property it can alternatively be shown that \( \models \varphi^e \rightarrow \varphi_1^e \) and \( \models \varphi^e \rightarrow \varphi_2^e \) hold individually.

In an asynchronous composition, the transitions of the components \( \mathcal{M}_1 \) and \( \mathcal{M}_2 \) may be triggered by two independent clocks and thus may react to changes in environment variables at different times. In particular, the transitions are not synchronized to the environment’s changes of the variables it controls. It is therefore necessary for \( \varphi_1^e \) and \( \varphi_2^e \) to be stutter invariant so that \( (P1) \) is sufficient in an asynchronous composition of \( \mathcal{M}_1 \) and \( \mathcal{M}_2 \).

The specifications \( \varphi_1^e \) and \( \varphi_2^e \) only allow to assume properties over the global environment variables \( \mathcal{X} \) and not over the other component in the composition. We therefore introduce the safety formulae \( \phi_1^s \in \mathcal{L}(\mathcal{X}_1) \) and \( \phi_2^s \in \mathcal{L}(\mathcal{X}_2) \), called refinement formulae. They are added to the local specifications and so the environment assumptions of \( \mathcal{M}_1 \) and \( \mathcal{M}_2 \) become \( \phi_1^s \land \varphi_1^e \in \mathcal{L}(\mathcal{X}_1) \) and \( \phi_2^s \land \varphi_2^e \in \mathcal{L}(\mathcal{X}_2) \) respectively.

The intuition behind separating these assumptions is that the refinements \( \phi_1^s \) and \( \phi_2^s \) can also contain variables from \( \mathcal{Y}_2 \) and \( \mathcal{Y}_1 \) respectively, while \( \varphi_1^e \) and \( \varphi_2^e \) can only contain variables from \( \mathcal{X}_1 \setminus \mathcal{Y}_2 \) and \( \mathcal{X}_2 \setminus \mathcal{Y}_1 \) respectively. The refinements will thus make environment assumptions about a component’s peers and will therefore somehow have to be satisfied by the system goals of the peers.
3.2. SPECIFYING DISTRIBUTED SYSTEMS

Let \( \varphi^i_1 \in \mathcal{L}(\mathcal{X}_1 \cup \mathcal{Y}_1) \) and \( \varphi^i_2 \in \mathcal{L}(\mathcal{X}_2 \cup \mathcal{Y}_2) \) be LTL formulae for the local guarantees. When these two specifications are satisfied by \( M_1 \) and \( M_2 \) respectively, they should together imply the global guarantee, that is \( \varphi \)

\[
(P2) \forall \sigma \in \mathcal{V}(\mathcal{V})^\infty. \sigma \models (\varphi^i_1 \land \varphi^i_2) \Rightarrow \sigma \models \varphi^e.
\]

We now also refine the guarantee part of the local specifications by the safety formulae \( \phi^e_1 \in \mathcal{L}(\mathcal{X}_1 \cup \mathcal{Y}_1) \) and \( \phi^e_2 \in \mathcal{L}(\mathcal{X}_2 \cup \mathcal{Y}_2) \). Thus the components \( M_1 \) and \( M_2 \) must satisfy their respective specifications:

\[
(P3) M_1 \models \varphi^i_1 \land \phi^e_1 \rightarrow \varphi^i_1 \land \phi^e_1 \text{ and } M_2 \models \varphi^i_2 \land \phi^e_2 \rightarrow \varphi^i_2 \land \phi^e_2
\]

Note again that \( \phi^e_1, \phi^e_2, \phi^i_1 \) and \( \phi^i_2 \) are safety formulae and \( \varphi^e, \phi^e_1, \phi^e_2 \) and \( \phi^i \) are stutter-invariant.

Circular Reasoning

Including the system variables of the peer component in the refinement formulae gives rise to circular dependencies: \( M_1 \) only has to guarantee \( \phi^i_1 \) if its environment assumptions \( \varphi^e_1 \land \varphi^i_1 \) are satisfied. This requires \( M_2 \) to satisfy \( \phi^i_2 \) which in turn only has to be guaranteed if \( \varphi^e_2 \land \varphi^i_2 \) is satisfied. The circularity comes from \( \phi^i_2 \) requiring \( \phi^i_1 \) to be satisfied by \( M_1 \) (or symmetrically, \( \phi^i_1 \) requiring \( \phi^i_2 \) to be satisfied by \( M_2 \)).

In order to establish sufficient conditions on the local specifications so that their composition satisfies the global specification, it is necessary to resolve this circularity. This is done by imposing the appropriate conditions on the refinement formulae \( \phi^i_1, \phi^i_2, \phi^e_1 \) and \( \phi^e_2 \). We require that for any sequence \( \sigma \) satisfying the local environment assumptions \( \varphi^e \), if a prefix can be extended to satisfy the local guarantee part of one component, then the same prefix and the next element in \( \sigma \) still must be able to be extended to satisfy the local environment assumptions of the other component.

It is important to require that the prefixes of sequences satisfying the global guarantees can be extended to satisfy the local guarantees, because the components must be prevented from entering a “one-way street” in their computations[3]. Consider for example a sequence \( s_0 s_1 s_2 \ldots = \sigma \models \varphi^e \) and a prefix \( s_0 s_1 \ldots s_n = \bar{\sigma} \in \text{pref}(\sigma) \) of \( \sigma \) that can be extended to satisfy \( \phi^i_1 \), the local guarantee refinement of the component \( M_1 \). That means that \( M_1 \) can make a finite number of transitions in order to satisfy \( \phi^i_1 \). If \( M_1 \) is currently in state \( q_n \) with \( \mathcal{V}(q_n) = s_n \) and it makes a transition \( q_n \xrightarrow{\rho_1} q_{n+1} \) s.t. \( \mathcal{V}(q_{n+1}) = s_{n+1} \) then the requirement that \( \bar{\sigma} \) extended by the element \( s_{n+1} \) in \( \sigma \) can be extended to satisfy \( \phi^i_1 \) simply means that by choosing to make this transition, \( M_1 \) actually makes such a transition in the “right direction”.

Also, the assumption must hold for one further element in the sequence, because it allows to prove properties about the transition from one state to the next in

---

[3] This is comparable to entering a tree-subgraph of the graph-representation of the transition relation, in which the leaf-nodes have no out-edges. Once a leaf of the tree is reached, no further transitions are possible.
3.2. SPECIFYING DISTRIBUTED SYSTEMS

a component without needing to assume anything about the next state of its peers. This resolves the circular reasoning. The resolution in the proof is based on induction, so for the base case the first element in $\sigma$ has to be able to be extended to satisfy the local guarantee parts of both components.

Bad Prefixes and Closure

The extensibility property introduced above can be formalized by defining the concept of bad prefixes, i.e. finite sequences that cannot be extended to satisfy a given formula.

**Definition 7.** A finite sequence of states $\sigma \in \mathcal{V}^+$ is a **bad prefix** of an LTL formula $\varphi \in \mathcal{L}(\mathcal{V})$ iff for all infinite sequences $\sigma \in \mathcal{V}^\infty$, the concatenation $\sigma\sigma$ does not satisfy $\varphi$, i.e. $\sigma\sigma \not\models \varphi$. The set of bad prefixes of $\varphi$ is denoted by $\text{badpref}(\varphi)$. Then for a sequence $\sigma \in \mathcal{V}^+$,

$$\sigma \not\in \text{badpref}(\varphi) \iff \exists \sigma \in \mathcal{V}^\infty. \sigma \models \varphi \land \sigma \in \text{pref}(\sigma).$$

We call a formula $\varphi_1$ **stronger** than another formula $\varphi_2$ iff $\models \varphi_1 \rightarrow \varphi_2$.

**Definition 8.** For an LTL formula $\varphi$, its **closure** $\mathcal{C}(\varphi)$ is the strongest safety formula implied by it. That is, $\mathcal{C}(\varphi)$ is the strongest safety property s.t. $\models \varphi \rightarrow \mathcal{C}(\varphi)$. \hfill \square

The sequences that satisfy $\mathcal{C}(\varphi)$ are exactly those sequences $\sigma$ s.t. each prefix of $\sigma$ is a prefix of some sequence satisfying $\varphi$. In other words, $\sigma$ satisfies $\mathcal{C}(\varphi)$ iff any of its prefixes can be extended to an infinite sequence satisfying $\varphi$. Note that

$$\sigma \models \mathcal{C}(\varphi) \iff \forall \sigma \in \text{pref}(\sigma). \sigma \not\in \text{badpref}(\varphi).$$

Jonsson and Tsay give a syntactic definition of closure in LTL using past operators [36]:

$$\mathcal{C}(\varphi) = \Box [\exists \bar{x} \in \mathcal{V}(\mathcal{X}). \square (\bar{x} = x) \land \Diamond (\varphi[\bar{x}/x])],$$

where $\varphi[\bar{x}/x]$ is the formula $\varphi$ with the values $\bar{x}$ substituted for the variables $x$, and the existential quantifier $\exists$ over finite sets $\mathcal{V}(\mathcal{X})$ is defined by

$$(\sigma, j) \models \exists \bar{x} \in \mathcal{V}(\mathcal{X}). \varphi \iff (\sigma, j) \models \bigvee_{\bar{x} \in \mathcal{V}(\mathcal{X})} \varphi[\bar{x}/x].$$

The existential quantification in the definition of the closure is over **all** valuations of the variables in the environment variables $\mathcal{X}$, or in other words, the **free** variables. This definition is used to prove the four-phase handshake protocol which is introduced below in Section 3.2.3.

Using the definition of bad prefixes, circular reasoning can be resolved if we impose the following restrictions on the refinement formulae:
(P4) For all sequences $s_0 s_1 s_2 \ldots \in V(X \cup Y)^\infty$ that satisfy the global environment assumptions $\varphi^e$, for any index $n \in \mathbb{N}_0$, if the prefix $s_0 s_1 \ldots s_n \not\in \text{badpref}(\phi_1^e)$ then also the prefix $s_0 s_1 \ldots s_n s_{n+1} \not\in \text{badpref}(\phi_2^e)$; similarly if the prefix $s_0 s_1 \ldots s_n \not\in \text{badpref}(\phi_2^e)$ then also the prefix $s_0 s_1 \ldots s_n s_{n+1} \not\in \text{badpref}(\phi_1^e)$; moreover, $s_0 \not\in \text{badpref}(\phi_1^e)$ and $s_0 \not\in \text{badpref}(\phi_2^e)$.

Feedback Interconnection Refinement Rule

We can now state the Feedback Interconnection Refinement Rule:

**Proposition 1.** ([67], Proposition 3) Given the architecture and the restrictions on the specifications outlined above, it holds that

$$
\begin{align*}
(M_1 \models \varphi^e \land \phi_1^e &\rightarrow \varphi_1^s \land \phi_1^s) \\
(M_2 \models \varphi^e \land \phi_2^e &\rightarrow \varphi_2^s \land \phi_2^s)
\end{align*}
$$

given (P1), (P2), (P4).

Proof sketch. Take a computation $\sigma \in \mathcal{L}_{M_1 \upharpoonright M_2}$ of $M_1 \upharpoonright M_2$. If $\sigma \not\models \varphi^e$, then the specification $\varphi^e \rightarrow \varphi^s$ is automatically met. So we assume that the environment assumptions $\varphi^e$ are satisfied. Then, by (P1), $\sigma$ also satisfies both local environment assumptions $\phi_1^e$ and $\phi_2^e$. Now (P4) ensures that the environment refinement formulae $\phi_1^e$ and $\phi_2^e$ are satisfied for the initial state and both components can proceed infinitely from this initial state (because of the no bad prefix property). Therefore, for the initial state, $\varphi_1^e \land \phi_1^e$ and $\varphi_2^e \land \phi_2^e$ are satisfied. By (P3), the local guarantees $\varphi_1^e \land \phi_1^e$ and $\varphi_1^e \land \phi_1^e$ are satisfied for the initial state as well. Then (P2) allows to conclude that $\varphi^s$ is satisfied by the initial state.

Because of (P4), the environment refinement formulae $\phi_1^e$ and $\phi_2^e$ are always satisfied for one further state than the system refinement formulae $\phi_1^s$ and $\phi_2^s$. Since $\varphi_1^s$ and $\varphi_2^s$ are satisfied by $\sigma$ anyway, $\sigma$ also satisfies $\varphi_1^s \land \phi_1^s$ and $\varphi_2^s \land \phi_2^s$, not just for the initial state. By a similar argument as for the initial state using (P3) and (P2) $\sigma$ satisfies $\varphi^s$. \hfill \blacksquare

Verifying the Hypotheses

In order to be able to use the Feedback Interconnection Refinement Rule, techniques are required to verify the hypotheses (P1)–(P4) that are required to deduce that the composition of the individually synthesized components satisfies the global specification.

The hypotheses give semantic conditions on the GR[1] specifications, which are harder to verify than syntactic conditions that are used e.g. in the work of Jonsson and Tsay [36]. The latter approach is not adopted because it requires the introduction of past operators and existential quantification into the temporal logic, which are not supported by the synthesis procedure based on JTLV.

Given $\varphi_1^e$, $\varphi_1^s$, $\varphi_2^e$ and $\varphi_2^s$, the properties (P1) and (P2) can be derived syntactically by using sequent calculus rules for LTL, see Section 2.1.3. The property (P3) is satisfied by construction from synthesis.
Verifying \([P4]\) in its full generality is more challenging. The simplest case is to just have \([\varphi e \land \phi_s^1 \rightarrow \phi^2] \) and \([\varphi e \land \phi_s^2 \rightarrow \phi^1] \), which can also be verified by sequent calculus.

### 3.2.2 Communication

For a coordinated action of the components to satisfy the global specification, it is often necessary to enable some form of information exchange [12]. One way to handle communication is proposed by Kloetzer and Belta [40, 41], but their approach does not involve the encoding of the communication protocols directly in LTL, allowing it to fit in the framework used in this thesis.

An approach more appropriate for our purposes is to define communication protocols directly in LTL [11, 23]. The refinement of the local specifications by safety formulae that can contain system variables of the peer component allows some limited form of information exchange.

However, in order to obtain reliable communication, it is required to introduce handshake protocols between the components. Their specification requires system variables to occur in the environment assumptions as well, which is not handled by the composition rule presented above. The stronger composition rules by Abadi and Lamport [2] and Jonsson and Tsay [36] support this but have the disadvantage that they introduce operators into LTL that the synthesis does not handle.

We will therefore identify where the Feedback Interconnection Refinement Rule fails to hold when system variables are allowed in the local environment assumptions and resolve them for the particular case of the four phase handshake protocol that is also used by Bloem et al. [11].

### Glitches

The symmetric situation to stuttering is when one FDS misses one or several transitions of its peers or the environment. When viewing shared variables

![Figure 3.2: Stuttering and Glitches](image-url)
3.2. SPECIFYING DISTRIBUTED SYSTEMS

as communication lines, then this may cause information to be lost and
environment assumptions to be violated, even though the environment behaves
appropriately. This is illustrated in Figure 3.2. The FDS \( M_1 \) is started in \( q_0 \)
while the environment is in state \( e_0 \). The transition of \( M_1 \) to \( q_1 \) follows after
only one transition of the environment to \( e_1 \), but \( M_1 \) observes \( M_2 \) to directly
transition from \( q_0^1 \) to \( q_2^2 \). This could cause problems if \( q_1^2 \) needs to be observed
in order to not violate the environment assumptions of \( M_1 \). Similarly, \( M_2 \) ob-
serves the environment to directly transition from \( e_1 \) to \( e_3 \).

These problems occur when the environment assumptions contain the next and
the eventually operator. In particular, environment assumptions like

\[
\square(x \rightarrow □ y),
\]

where \( x \) and \( y \) are system and environment variables respectively, are not per-
missible. No environment can guarantee that it will have asserted \( y \) the next
time an FDS is observing it. This is because there can be a delay between \( x \) be-
ing asserted and this information reaching the environment. The above formula
is neither stutter nor glitch invariant. Also,

\[
\square(x \rightarrow □ y)
\]

is not glitch invariant, even though it is stutter invariant. The problem is that
the environment may satisfy this assumption by asserting \( y \) for only one step
and then clearing it again. This might be missed by an FDS that makes this
environment assumption.

It is therefore important to formulate specifications to be robust to such glitches.
If a finite number of transitions are missed, correct operation must still be
guaranteed.

3.2.3 The Four-Phase Handshake Protocol

Consider the case of transmitting an integer \( \delta \in [0, n) \) from a sender \( M_S \) to
a receiver \( M_R \). The sender controls a request signal \( r \in [\epsilon, n) \). If \( r = \epsilon \), this
indicates that no data should be transmitted. If \( r \neq \epsilon \), then the data that is
to be transmitted is \( \delta = r \). The receiver controls a boolean acknowledgement
signal \( a \). The request signal \( r \) appears as an environment variable \( r \in [\epsilon, n) \)
at the receiver. Similarly, the acknowledgement signal \( a \) appears as a boolean
environment variable \( a \) at the sender.
3.2. SPECIFYING DISTRIBUTED SYSTEMS

Thus, the architecture that contains the sender $M_S$ and the receiver $M_R$ can be visualized as in Figure 3.4. Comparing this with the generic composite architecture in Figure 2.9 yields $E_S = \{t\}$, $S_S = \emptyset$, $T_{R,S} = \{r\}$, $T_{S,R} = \{a\}$, $E_R = \emptyset$ and $S_R = \{s\}$. Thus the composite FDS $M_S \| M_R$ has the environment variables $X = \{t\}$ and the system variables $Y = \{a,r,s\}$. When using the protocol in a larger specification, the trigger $t$ is likely to also be a system variable of $M_S$.

The four-phase handshake protocol can be informally described as follows. The sender $M_S$ initiates a transfer by setting $r = \delta$. Once $M_R$ detects this, it may use the transferred data somehow, and then raises $a$. When $M_S$ sees that $a$ has been asserted, it resets $r = \epsilon$. After $M_R$ detects this, it clears $a$, completing the transmission of $\delta$. This is outlined in Figure 3.3.

\section*{Sender Specification}

The sender assumes one liveness and two safety properties about the receiver. It is assumed that if the request is reset, the acknowledgement will be cleared eventually, allowing for a new request to be made:

\begin{enumerate}
  \item $\square(r = \epsilon \rightarrow \Diamond \neg a)$.
\end{enumerate}

The assumed safety properties assert that the acknowledgement signal does not change in a way that is not allowed by the protocol. There is no acknowledgement without a request:

\begin{enumerate}
  \item $\square(\neg a \land \Diamond a \rightarrow r \neq \epsilon)$.
\end{enumerate}

Also, the acknowledgement may only be cleared if the request has been reset before:

\begin{enumerate}
  \item $\square(a \land \Diamond \neg a \rightarrow r = \epsilon)$.
\end{enumerate}

The sender guarantees three safety properties. No liveness property is guaranteed, because the sender may just remain idle. However, a system using this protocol will have to contain a liveness property that implicitly guarantees liveness in the protocol. Also, as long as the request is not acknowledged, it must stay at the same value:

\begin{enumerate}
  \item $\bigwedge_{i \in \{0,n\}} \square(\neg a \land r = i \rightarrow \Diamond (r = i))$.
\end{enumerate}

The request may only be reset to $\epsilon$:

\begin{enumerate}
  \item $\bigwedge_{i \in \{0,n\}} \square(r = i \land \Diamond (r \neq i) \rightarrow \Diamond (r = \epsilon))$.
\end{enumerate}
Lastly, the sender responds to an acknowledgement by resetting the request:

\[ \text{IX } \Box(a \rightarrow \Diamond(r = \epsilon)). \]

Note that this last guarantee also ensures that the sender may only initiate a transmission if the acknowledgement signal is low. This could be expressed by the additional guarantee \( \Box(r = \epsilon \land \Diamond(r \neq \epsilon) \rightarrow \neg a) \), but clearly this is already implied by IX.

**Receiver Specification**

On the receiver side, one liveness and two safety assumptions are made. The receiver assumes that each acknowledgement eventually results in the request to be cleared:

\[ \text{IV } \Box(a \rightarrow \Diamond(r = \epsilon)). \]

It also assumes that the request reacts appropriately to the acknowledgement. Similar to guarantee VII, no transmission is initiated if the acknowledgement is still high:

\[ \text{V } \Box(r = \epsilon \land \Diamond(r \neq \epsilon) \rightarrow \neg a). \]

Also, the request may only be cleared if it has been acknowledged:

\[ \text{VI } \Box(r \neq \epsilon \land \Diamond(r = \epsilon) \rightarrow a). \]

The receiver guarantees that each request will be acknowledged:

\[ \text{XI } \Box(r \neq \epsilon \rightarrow \Diamond a). \]

And it guarantees that the resetting of the request results in the acknowledgement to be cleared:

\[ \text{XII } \Box(r = \epsilon \rightarrow \Diamond \neg a). \]

Moreover, the acknowledgement may only be raised on a request:

\[ \text{XIII } \Box(\neg a \land \Diamond a \rightarrow r \neq \epsilon). \]

Lastly, the acknowledgement may not be cleared unless the request is reset:

\[ \text{XIV } \Box(a \land \Diamond \neg a \rightarrow r = \epsilon). \]

**Reliability of the Four-Way Handshake Protocol**

It now remains to show that the four-phase protocol reliably transmits data from the sender to the receiver. To show this, we introduce a boolean variable \( t \) into the environment variables of the sender \( M_S \), which will act as a “trigger” to a message being sent. The data will just be a single constant \( \delta \in [0,n) \). Similarly, introduce an integer variable \( s \in [0,n) \) into the system variables of the receiver, which will act as a “sink” where the sent data will be stored for later use. We add the response property

\[ \text{X } \Box(t \rightarrow \Diamond(r = \delta)) \]
3.2. SPECIFYING DISTRIBUTED SYSTEMS

to the guarantees of the sender, which ensures that the data will eventually be sent, once the transfer has been triggered by “\( t = \text{True} \)”. Similarly, the response property

\[
\text{XV } \Box(r = \delta \rightarrow \Diamond(s = \delta))
\]

will be added to the guarantees of the receiver. Note that while in XV the eventually operator might be substituted by a next operator, this cannot be done in X, since changing \( r \) is restricted by \( a \), which is not controlled by \( M \).

The full local specifications are shown in Figure 3.5. The global specification is

\[
\Box(t \rightarrow \Diamond(s = \delta)),
\]

requiring that once a transmission is triggered by \( t \) being asserted, the data \( \delta \) eventually arrives correctly at the sink \( s \). We thus want to prove the following:

**Proposition 2.** If \( M_S \models \varphi^e_S \rightarrow \varphi^S_S \) and \( M_R \models \varphi^e_R \rightarrow \varphi^S_R \), then \( M_S | M_R \models \Box(t \rightarrow \Diamond(s = \delta)) \).

**Proof.** We make use of the Composition Theorem of Abadi and Lamport ([2], Theorem 3), which in our particular case is:

\[
\begin{align*}
\text{(A1)} & \quad \models \mathcal{C}(\varphi^e_S) \land \mathcal{C}(\varphi^e_R) \rightarrow \varphi^S_S \land \varphi^S_R \\
\text{(A2)} & \quad \models \mathcal{C}(\varphi^e_S) \land \mathcal{C}(\varphi^e_R) \rightarrow \mathcal{C}(\Box(t \rightarrow \Diamond(s = \delta))) \\
\text{(A3)} & \quad \models \varphi^S_S \land \varphi^S_R \rightarrow \Box(t \rightarrow \Diamond(s = \delta))
\end{align*}
\]

(3.1)

Note that we only use a nesting depth of the next operator of one in the GR[1] specifications, so the explicit translation to TLA is omitted. For the closure required for the Composition Theorem, the above LTL definition by Jonsson and Tsay will be used.

Precondition (A3) can be established by taking X \( \Box(t \rightarrow \Diamond(r = \delta)) \) and XV \( \Box(r = \delta \rightarrow \Diamond(s = \delta)) \). From these specifications it follows that \( \Box(t \rightarrow \Diamond(s = \delta)) \). This is intuitively clear, but can also established by a sequent calculus proof.

For the preconditions (A1) and (A2), consider the closure of \( \varphi^S_S \), which is \( \Box(\exists \bar{t}, \bar{a}. \Box(\bar{t} = t \land \bar{a} = a)) \land \Box(\varphi^S_S[\bar{t}/t][\bar{a}/a]) \) and the closure of \( \varphi^S_R \), which is \( \Box(\exists \bar{r}. \Box(\bar{r} = r)) \land \Box(\varphi^S_R[\bar{r}/\bar{r}]) \). We consider the individual cases for values of \( \bar{t} \), \( \bar{a} \) and \( \bar{r} \) to deal with the existential quantification, which, by definition, is just a shorthand for disjunction.

Note that \( \models \Box \Box \varphi \leftrightarrow \Box \varphi \) and \( \models \Box \Box \varphi \leftrightarrow \Box \varphi \) for any LTL formula \( \varphi \), which helps to simplify the closures of the formulae we are considering.

**Case 1.** \( \bar{t} = \text{True}, \bar{a} = \text{True} \):

Evaluating the conjuncts for these valuations yields \( \Box \Diamond(r = \epsilon) \) from IX and \( \Box \Diamond(r = \delta) \) from X. These two formulae can never hold together and thus in this case the closure of \( \varphi^S_S \) is simply false. As false implies everything, both (A1) and (A2) hold in this case.
3.2. SPECIFYING DISTRIBUTED SYSTEMS

\[ M_S \models \begin{cases} 
    I & \Box(r = \epsilon \rightarrow \Diamond \neg a) \land \\
    II & \Box(\neg a \land \Box a \rightarrow r \neq \epsilon) \land \\
    III & \Box(a \land \Box \neg a \rightarrow r = \epsilon) 
\end{cases} \rightarrow \begin{cases} 
    \Box(r = \epsilon \land \Box(r \neq \epsilon) \rightarrow \neg a) \land \\
    \Box(\neg a \land r = i \rightarrow \Box(r = i)) \land \\
    \Box(r = i \land \Box(r \neq i) \rightarrow \Box(r = \epsilon)) \land \\
    \Box(a \rightarrow \Box(r = \epsilon)) \land \\
    \Box(t \rightarrow \Diamond(s = \delta))
\end{cases} \]

\[ M_R \models \begin{cases} 
    IV & \Box(a \rightarrow \Diamond(r = \epsilon)) \land \\
    V  & \Box(r = \epsilon \land \Box(r \neq \epsilon) \rightarrow \neg a) \land \\
    VI & \Box(r \neq \epsilon \land \Box(r = \epsilon) \rightarrow a)
\end{cases} \rightarrow \begin{cases} 
    \Box(r \neq \epsilon \rightarrow \Box a) \land \\
    \Box(r = \epsilon \rightarrow \Box \neg a) \land \\
    \Box(\neg a \land \Box a \rightarrow r \neq \epsilon) \land \\
    \Box(a \land \Box \neg a \rightarrow r = \epsilon) \land \\
    \Box(r = \delta \rightarrow \Diamond(s = \delta))
\end{cases} \]

Figure 3.5: Specification of the four-phase handshake protocol.
3.2. SPECIFYING DISTRIBUTED SYSTEMS

Case 2. \( t = \text{True}, \ a = \text{False}, \ r = \epsilon \):

Here the conjunct resulting from X is \( \Box \Diamond (r = \delta) \), but we also have \( \Box(r = \epsilon) \) due to the existential quantification. Thus, again the antecedent is false and so both (A1) and (A2) are implied.

Case 3. \( t = \text{True}, \ a = \text{False}, \ r = \delta \):

Evaluating the conjuncts in the closures for these variables yields the conjunct \( \Box \Diamond (s = \delta) \) for XV. Moreover, the conjuncts \( \Box(t = \text{True}), \ \Box(a = \text{False}) \) and \( \Box(r = \delta) \) are in the closures from the existential quantification.

For (A1) we need to deduce I–VI from these conjuncts. I and VI follow from \( \Box(r = \delta) \) and II–V follow from \( \Box(a = \text{False}) \).

For (A2) we need to verify the individual terms of the closure of \( \Box(t \rightarrow \Diamond (s = \delta)) \), which is \( \Box(\exists \bar{t}. \Box(t = \text{True}) \land \Diamond (s = \delta)) \).

In order to resolve the existential quantification, we again consider both truth values of \( t \). However, since we are now proving the closure as a consequent, we only need to exploit one value of \( t \) for which the closure of \( \Box(t \rightarrow \Diamond (s = \delta)) \) is implied by the antecedents.

Let \( t = \text{True} \). Then the closure becomes \( \Box(\exists \bar{t}. \Box(t = \text{True}) \land \Diamond (s = \delta)) \). The second conjunct follows simply from \( \Box(t = \text{True}) \) from the closure of \( \varphi^s \). The first conjunct follows from XV, i.e. \( \Box(s = \delta) \).

Case 4. \( t = \text{True}, \ r \neq \epsilon \) and \( r \neq \delta \):

Evaluating the conjuncts for these variables yields the conjuncts \( \Box(a \lor \Diamond \neg a) \) from XIII and \( \Box(\neg a \lor \Diamond a) \) from XIV. These two conjuncts already yield a contradiction, and so (A1) and (A2) are both satisfied.

Case 5. \( t = \text{False}, \ a = \text{True}, \ r = \epsilon \):

The conjunct in the closure resulting from XII is \( \Box \Diamond a \). This stands in contradiction with the conjunct \( \Box(a = \text{True}) \) from the existential quantification. Thus both (A1) and (A2) are satisfied.

Case 6. \( t = \text{False}, \ a = \text{True}, \ r \neq \epsilon \):

The conjunct in the closure resulting from X is \( \Box \Diamond (r = \epsilon) \), but the existential quantification requires \( \Box(r \neq \epsilon) \). This leads to an immediate contradiction, so both (A1) and (A2) are satisfied.

Case 7. \( t = \text{False}, \ a = \text{False}, \ r = \epsilon \):

To prove (A1) we again need to be able to deduce I–VI from the conjuncts of the closure. I–IV follow from \( \Box(a = \text{False}) \) while V and VI follow from \( \Box(r = \epsilon) \).
3.2. SPECIFYING DISTRIBUTED SYSTEMS

For (A2) consider again the closure $\mathcal{E}(\Box(t \rightarrow \Diamond(s = \delta))) = \Box(\exists \bar{t}, \Box(t = \bar{t}) \land \Diamond(t \rightarrow \Diamond(s = \delta)))$. By taking $\bar{t} = \text{False}$, this closure is implied simply by $\Box(t = \text{False})$.

**Case 8.** $\bar{t} = \text{False}$, $\bar{a} = \text{False}$, $\bar{r} = \delta$:

For the preconditions (A1) and (A2) the conjuncts from the existential quantification are sufficient: I–V follow from $\Box(a = \text{False})$ and VI follows from $\Box(r = \delta)$. Moreover, in the same way as in Case 6, (A2) follows from $\Box(t = \text{False})$.

This covers all cases and so the preconditions (A1) and (A2) have been established. The result follows from the Composition Theorem (3.1) stated at the beginning of this proof. □

Note that we had to use (3.1), a stronger result than the Feedback Interconnection Refinement Rule in order to be able to incorporate the system variables in the environment assumptions, and to include liveness formulae in the specifications.

When using the four-phase handshake protocol in the form just proved several things should be noted. It is always possible to use weaker assumptions, as long as the specification remains synthesizable. In particular, omitting conjuncts in $\varphi_{S}$ or $\varphi_{R}$ does not change the correctness of the protocol. However, the guarantees $\varphi_{S}$ and $\varphi_{R}$ must be provided in a specification. They need not occur in the same syntactic form, but can be strengthened to an arbitrary degree without affecting correctness of the protocol.
3.2. SPECIFYING DISTRIBUTED SYSTEMS
Chapter 4

Searching

One of the main challenges in SAR is to locate the targets that need to be rescued. In order to avoid ambiguities, the physical environment of the robot, such as a building in USAR or a landscape in WiSAR, is referred to as topology. The controllers are specified and synthesized with a two-player game in mind, which then can be thought of as taking place on a graph representing the topology. Assuming that a SAR robot can detect the presence of a target if it is in the same vertex of the graph-representation of the topology, the problem is to detect the vertex the target is in.

The solution depends on the capabilities of the robots and the targets, as well as the amount of information they have. Here we differentiate searching for a stationary and a moving target. When the target is stationary, it is sufficient to ensure that each vertex in the graph is eventually visited by at least one robot. When the target can move, we formulate a pursuit-evasion game, in which an evader (the target) tries to avoid being captured by one or more pursuers (the robots). The controller specifications are developed from the winning strategies for the cops of this game.

4.1 Preliminaries

Before presenting the search strategies for the different scenarios, we make precise the concept of the graph representation of the topology and the way sensors are incorporated in our specification framework from Section 3.

4.1.1 Graph Representation of the Topology

In order to reason about the movement of the robots in an SAR task, the topology must be represented internally in a way that admits LTL specifications about the positions of the robots. The topology is partitioned into discrete “cells” that are typically polygonal shapes. The robots move from one cell

\footnotesize{This can be extended to require any number of robots to visit the vertex at the same time.}
4.1. PRELIMINARIES

Figure 4.1: Graph representing a topology with five cells. Each vertex stands for a cell and the arrows are the allowable moves between cells, which can be thought of e.g. as doors or corridors between rooms in a building. The graph implicitly contains a self-loop for each vertex.

The changes of cellID are constrained by the topology, which can be represented by a directed graph $G = (V_G, E_G)$. Each vertex in $V_G = \{v_0, v_1, \ldots, v_{n-1}\}$ corresponds to a cell. The possible transitions between the cells are the edges $E_G$.

An edge $e$ going from the vertex $u \in V_G$ to the vertex $v \in V_G$ is an ordered pair $(u, v) \in V_G \times V_G$. The edge $e = (u, v)$ is an in-edge of $v$ and an out-edge of $u$.

Definition 9. A path is a sequence of vertices $p = v_0v_1\ldots v_{l-1}$ over $V_G$ s.t. $(v^i, v^{i+1}) \in E_G$ is an edge for all indices $i \in [0, l-1)$. A simple path is a path with unique vertices, i.e. it contains no cycles. The $i^{th}$ element of the path $p$ is also denoted by $p(i) = v^i$, and we say that $p$ is a path from $p(0)$ to $p(l-1) = p(|p| - 1)$.

Definition 10. A graph $G = (V_G, E_G)$ is strongly connected iff there exists a path from every vertex $v \in V_G$ to every other vertex $u \in V_G$. Denote the set of all strongly connected directed graphs with $n$ vertices by $\mathcal{S}_n$.

We require the graph $G$ representing the topology to be strongly connected, that is, it is possible to get from every vertex to any other vertex of the graph. If a graph is not strongly connected, there may be cells that the robots are not able to access, depending on their initial positions. Targets located in such inaccessible cells could never be found with any strategy.

The constraints on cellID are incorporated in the guarantee part of a robot’s GR[1] specification. Consider for example the graph shown in Figure 4.1 which is strongly connected and therefore qualifies as a valid basis for synthesis. A robot is always allowed to stay at a vertex, so the self loops are included implicitly. For notational convenience, we use the boolean variables $X_i$ instead of explicitly writing $cellID = i$, i.e. $X_i = True \iff cellID = i$. Further let
4.1. PRELIMINARIES

Let $n = |V_G|$ be the number of vertices in $G$. Then, for the graph in Figure 4.1, the corresponding specification is

$$\varphi_G = \Box (X_0 \rightarrow \Box (X_0 \lor X_1 \lor X_3)) \land$$

$$\Box (X_1 \rightarrow \Box (X_0 \lor X_1)) \land$$

$$\Box (X_2 \rightarrow \Box (X_0 \lor X_2 \lor X_4)) \land$$

$$\Box (X_3 \rightarrow \Box (X_1 \lor X_2 \lor X_3)) \land$$

$$\Box (X_4 \rightarrow \Box (X_2 \lor X_4)) \land$$

$$\Box (\bigvee_{h \in [0,n]} (X_h \land \bigwedge_{i \in [0,n] \setminus \{h\}} \neg X_i)), $$

where the last line is a mutual exclusion formula, preventing the robot to think that it is in two cells at the same time. This ensures that $\Box(\text{cellID} = i \leftrightarrow X_i)$.

4.1.2 Generating Random Strongly Connected Graphs

The methods developed in this chapter are tested on randomly generated graphs. We use the Erdős-Rényi Model in order to generate random directed graphs that are strongly connected [24].

For this purpose, we represent a graph $G = (V_G, E_G)$ with $n$ vertices $V_G = \{v_0, v_1, \ldots, v_{n-1}\}$ by its adjacency matrix $\text{Adj}(G) \in \mathbb{Z}^n \times \mathbb{Z}^n$, where the $i, j^{th}$ entry of $\text{Adj}(G)$ is nonzero exactly if there is an edge $(v_i, v_j) \in E_G$ from $v_i$ to $v_j$. We require that a graph contains all self-loops, so the diagonal of $\text{Adj}(G)$ for any graph $G$ that we are considering contains only nonzero elements. Since any nonzero entry in $\text{Adj}(G)$ represents an edge, the adjacency matrix can also be used to store state information of edges by using different nonzero values. This is used for example for the moving target search, see Section 4.4.1.

To generate a random graph $G$ with $n$ vertices, we select a constant $P$ such that

$$P > \frac{1}{n - 1}.$$ 

Further, let $X$ be a continuous random variable that is uniformly distributed over the range $\{x \mid 0 \leq x < 1\} \subseteq \mathbb{R}$. Next, we draw one sample for each pair of vertices from $X$, yielding the $n^2$ samples $e_{1,1}, e_{1,2}, \ldots, e_{1,n}, e_{2,1}, e_{2,2}, \ldots, e_{n,n}$. For all $i$ and $j$, if $e_{i,j} < P$ then let the $i,j^{th}$ element of $\text{Adj}(G)$ be one and zero otherwise. Thus, a matrix of ones and zeros is generated that defines the graph $G$.

With this choice of $P$, the resulting graph $G$ almost surely contains one “giant” component, i.e. a subset of vertices covering almost the entire graph in which each vertex is reachable by a path from each other vertex in the same set. However, we still need to check, whether $G$ is indeed a strongly connected graph. This can be done by self-multiplying $\text{Adj}(G)$ $n$ times and testing whether or not all elements in $\text{Adj}(G)$ are nonzero. The process of generating a random graph is repeated until this test yields a positive result, confirming that the graph is strongly connected.
4.2. STATIONARY TARGET SEARCH

4.1.3 Sensors

So far, the input variables of an FDS were considered to be controlled by an abstract environment. When an FDS is executed on a microprocessor, the input variables are abstractions e.g. from inputs received from sensors or data transfer lines connected to the processor. For example, a boolean environment variable may be the abstraction of the input received from a sensor detecting the state of a switch.

When considering an FDS to be the controller of a robot that moves on a graph $G$, sensors can be differentiated between gathering information from the entire graph or just a subgraph of $G$. In the former case we speak of **global** information, while information from a subgraph in the vicinity of the robot is called **local**. In the extreme case, a robot may only obtain sensor inputs from the vertex it is on.

Thus, sensor inputs, e.g. represented by an environment variable $s$, depends on the system variable $cellID$. However, the robot does not know about this dependency. Therefore, when simulating the FDS, we must ensure that the state of the graph is maintained globally, but only the appropriate local information is available to the robot by changing $s$ accordingly. This is further discussed in Section 4.5.2 and is implemented through connectors as explained in Section 5.1.1.

4.2 Stationary Target Search

In an USAR application, taking place for example in a collapsed building, the targets are usually assumed to be stationary, and thus the search only involves visiting each vertex by a robot until the target is detected. A global specification therefore requires that once a vertex contains a target, it is eventually visited by at least one robot. Here we present the concept of stationary target search by considering only a single target at a time in the graph. In the implementation presented in Section 5.1 several targets can be rescued at the same time.

4.2.1 Global Specifications

Let $G = (V_G, E_G)$ be a directed graph with $n = |V_G|$ vertices. In order to formulate the global specification, we introduce the global integer environment variable $t \in [\epsilon, n)$, which indicates whether and where a target needs to be rescued. If $t = \epsilon$ then no target is in the graph, but if $t \neq \epsilon$ then a target is located on vertex $v_t$, i.e. the vertex with index $t$. This represents the actual state of the target, but the robots have no direct information about the value of $t$. Rather, the robots are equipped with sensors that provide local information about the vertex the robot is currently on. Thus the robot only sees the value of $t$ when it is on the vertex $v_t$.

Assume there are $m$ robots $\{R_0, R_1, \ldots, R_{m-1}\}$ searching for the targets, each with a unique index $j \in [0, m)$. The position of robot $R_j$ is controlled via the
system variable \(cellID_j \in [0,n]\). For notational convenience, we use the boolean variable \(X^i_j\) instead of explicitly writing \(cellID_j = i\).

The main global system goal therefore can be stated as follows:

\[
\varphi_s = \bigwedge_{i \in [0,n]} \Box(t = i \rightarrow \Diamond(\bigvee_{j \in [0,m]} X^i_j)).
\] (4.1)

This expresses that once a target is at vertex \(v_i\), at least one robot will find it, i.e. at least one robot will eventually move to the vertex \(v_i\). In order for this to be realizable, we have to at least assume that the target stays at the vertex until it is discovered. This is the **stationarity assumption**, and it can be expressed by the global environment assumption

\[
\varphi^e = \bigwedge_{i \in [0,n]} \Box(t = i \land \Diamond(t \neq i) \rightarrow \bigvee_{j \in [0,m]} X^i_j),
\]

which says that in order for \(v_t\) to change from a value that is not \(\epsilon\), at least one robot has to be at the target’s vertex. A different way of expressing the global system goal is

\[
\hat{\varphi}^s = \Box(t \neq \epsilon \rightarrow \Diamond(t = \epsilon)),
\]

which makes the actions required for the target at \(t\) to be rescued implicit. In order for \(\hat{\varphi}^s\) to be realizable, the environment assumptions must indicate how \(t\) reacts to the actions for the robots, i.e. how the system variables can be changed in order to clear \(t\). For example, \(t\) can be cleared if a robot moves to the vertex \(v_t\), which is expressed by the environment assumption

\[
\hat{\varphi}^e = \bigwedge_{i \in [0,n]} \Box(t = i \land \bigvee_{j \in [0,m]} X^i_j \rightarrow \Diamond(t = \epsilon)).
\]

Note that the two specifications here might not be synthesizable and not even realizable. They are merely shown to illustrate the principle behind specifying stationary target search globally.

### 4.2.2 Compositional Specifications

The specification \(\varphi^e \rightarrow \varphi^s\) (or \(\hat{\varphi}^e \rightarrow \hat{\varphi}^s\)) might not be complete or realizable, but it expresses the main idea behind globally specifying a distributed search in a graph. If a synthesis algorithm existed for general compositional synthesis, this specification could directly be used to synthesise a reactive controller for each robot in the architecture, s.t. together the robots would implement the global specification. However, since no such algorithm exists, the specification must be decomposed manually.

**Minimal Specification**

Each robot can have a simple local specification

\[
\bigwedge_{i \in [0,n]} \Box \Diamond X^i_j,
\]

59
4.2. STATIONARY TARGET SEARCH

stated here for the robot $R_j$. Note that the robot knows nothing about the location $t$ of the target and no sensor information is used. This way of searching assumes that the target reacts automatically to the robot being in the same cell by becoming rescued. Note however that in order to actively engage in the rescue of a target, the robot needs to react to a sensor input that indicates that it is in the target’s cell.

In this specification, there is no coordination between the robots, and so the search might be inefficient. However, we are not so much concerned about efficiency of our controllers as with their correctness. Synthesizing a controller from this specification would indeed satisfy the global specification given above. However, when not one, but several robots are required to find a target, a different approach is needed. With stationary targets this is not hard to implement, since a robot can just stay with the target until a second robot arrives, requiring the robot to have information from a sensor that it is in the target’s cell.

Circular Search

Another approach is to explicitly encode the paths that the robots have to take through the graph in order to visit each vertex at least once. This again does not allow for coordination between the robots. It has however one decisive advantage over the previous approach: It is explicitly known to the robot when the search is finished. In the previous approach, only the finiteness of the state space of the synthesized FDS induces a bound on the time to complete searching the graph.

The paths are extracted from the graph by finding a shortest cycle that contains all vertices, which can be done by the algorithm FindCycle in Table 4.1. The algorithm is provided with a starting vertex $u$ and returns a shortest path $p$ that starts at $u$ and ends at a vertex $v$, such that there exists an edge $(v, u) \in E_G$ and $p$ contains each vertex in $V_G$ at least once.

In the specification, the robot can explicitly be instructed to follow the path $p$ returned by FindCycle. Since the cycle formed by the edges connecting the vertices in $p$ might not be Hamiltonian, it is not sufficient to just encode this with guarantees such as $\Box (X_{p(0)}^{j} \rightarrow \bigcirc X_{p(1)}^{j})$. This is because some vertices might need to be visited several times and at such vertices the choice of the next vertex depends on the current position in the path. By introducing a counter system variable $c \in [0, |p|)$ and letting the robot visit the elements on the path in order it is possible to keep track of the elements in $p$ already visited.

The search is started with the counter at zero and the robot being at $p(0) = u$:

$$X_{p(0)}^{j} \land c = 0.$$

\footnote{Arbitrary efficiency bounds cannot be provided in GR[1] specifications that have a limited nesting depth of the next operator $\bigcirc$ and lack the bounded until operator $U^{\leq k}$. In fact, the allowed nesting depth of $\bigcirc$ is just one in GR[1] specifications.}

\footnote{We assume that the topology of the environment does not change, so it is a valid solution to explicitly encode a path for the robot. However, if the topology is unknown or changing, this approach is no longer appropriate.}

\footnote{A cycle is Hamiltonian if it contains all vertices of a graph exactly once.}
4.2. STATIONARY TARGET SEARCH

FindCycle \((u)\)
paths = [[u]]
while True do
  extendedPaths = []
  for p in paths do
    if p does not contain all vertices in \(V_G\) or p does not end in \(u\) then
      newpaths = ExtendByOne(p)
      append newpaths to extendedpaths
    else
      take away the last element of p (i.e. p no longer ends in \(u\))
      return p
  paths = extendedpaths

ExtendByOne \((p)\)
if p is empty return
paths = []
for v ∈ \(V_G\) do
  if \(p(|p| - 1), v\) ∈ \(E_G\) then
    newpath = p extended with v
    append newpath to paths
return paths

Table 4.1: Algorithm to find cycles containing all vertices in a graph. Calling FindCycle\((u)\) on some starting vertex \(u \in V_G\) yields a list of vertices \(p\) that induces a cycle in \(G\). Assuming list operations to be \(O(1)\), the runtime complexity of this naïve implementation of FindCycle is \(O(n \cdot n!)\) for dense graphs, and \(O(n^4)\) for sparse graphs with \(O(|E_G|) \approx O(|V_G|)\).
4.3. MOVING TARGET SEARCH

Note that \( X^j_{p(0)} \) is a synonym for \( cellID_j = p(0) \), so all variables \( X^j_i \) with \( i \neq p(0) \) necessarily evaluate to “False”. The counter is restricted to only increase, unless it has to be reset to zero in order to loop when the path is completed. Define \( nextval(\gamma) = (\gamma + 1 + |p|) \mod |p| \) to be the next value of the counter when the current value is \( \gamma \), where \( \mod \) is the infix operator for the remainder of integer division. Then this rule can be written as:

\[
\Box \left( \bigvee_{\gamma \in [0,|p|]} (c = \gamma \land \neg (c = nextval(\gamma))) \right).
\] (4.2)

The robot moves along the path, increasing the counter by one each step:

\[
\bigwedge_{\gamma \in [0,|p|]} \Box (X^j_{p(\gamma)} \land c = \gamma \rightarrow \neg (X^j_{p(nextval(\gamma))} \land c = nextval(\gamma))).
\] (4.3)

Both (4.2) and (4.3) are required in a complete specification: (4.2) prevents the counter to change to a value not corresponding to a valid position in the path (safety property), which would allow the robot to just remain at the same vertex. (4.3) is required for the robot to make any progress at all (liveness property), forcing the counter to increase.

This way of searching a graph is used in the implementation of moving target search, since there it is necessary that the robot knows when the search is finished, which is indicated by \( c = |p| - 1 \). A full specification of this type of search is given in Section 5.2 when searching for particular vertices in the topology.

Other Search Methods

Searching a graph can also be implemented as a depth-first-search or as another exploration algorithm. The specification of such algorithms is not straightforward, but can be done using counters and storing information in the vertices similar to the approach presented in Section 4.5 under the heading “Global Storage”, which ensures that the state space does not explode.

A depth-first search can handle an unknown environment topology of finite extent, if the vertices are able to store information for the robot and provide the robot with information of its neighboring vertices. However, synthesis from an LTL specification might not be the best approach to implement such an algorithm that has already been proved to work in a standard implementation.

4.3 Moving Target Search

When the target is moving, such as in a WiSAR application, it is no longer sufficient to visit each vertex eventually, like in the stationary target search. For most topologies, a fixed number of robots might never be able to find a moving target. Consider for example the situation in Figure 4.2, where a single robot (●) is trying to find a moving target (○) by moving on the same vertex in the topology. The target can evade the robot by always going to the opposite vertex.
at the same time as the robot moves. If the target knows the robot’s strategy, it can always evade the robot.

It is therefore necessary to implement a search strategy that ensures that even a moving target cannot escape the robots. This problem has been considered before and successfully implemented, for example in the work of Vidal et al. [88]. These implementations use probabilistic methods to find the target; we use a different approach in this thesis, and consider the problem of finding a winning strategy for a game of the robots against the targets on the graph representing the topology.

### 4.3.1 Game Theoretic Approach

Trying to find a moving target can be seen as a form of a pursuit-evasion game in which the target tries to evade its pursuers, the SAR robots. The objective of the game in its original formulation is to find the minimum number of robots necessary to guarantee that the target is eventually found. Depending on what moves are allowed for the robots and the targets, the robots must implement different strategies in order to reliably find the target. A recent survey by Chung et al. provides a taxonomy of properties for pursuit-evasion games [17].

**Cops and Robbers Game**

We consider a particular form of pursuit-evasion game called the cops and robbers game first introduced by Nowakowski and Winkler [66]. A number of cops and robbers take turns in moving on a directed graph by traversing edges from vertex to vertex. A cop captures a robber if they find themselves on the same vertex or edge. Cops can move one edge at a time, but the robbers can move with infinite speed, i.e. in each turn they can move along any number of edges. However, they cannot pass through vertices that are occupied by cops. To align with standard formulations, a robber may hide either on a vertex or on an edge.

The cops **win** the game if the robber is eventually captured, while the cops **lose** the game if any of the robbers are able to evade the cops forever. The robbers are supposed to be adverse and always choose the worst possible move against the cops. They also know exactly where the cops are and their strategies, while the cops have no information on the position or strategy of a robber, unless it is
captured. The game is considered as a two-player game between $C$, the central coordinating authority of the cops against the adverse robbers, modelled as a single player $R$.

A *strategy* for the cops defines the moves of the cops on the graph depending on the observations about the targets. A *winning strategy* for the cops is a strategy that guarantees that all robbers are found in a finite number of moves. The game is *solved* by finding the minimum number of cops for which a winning strategy exists for a given graph.

Note that the terms robot, cop and pursuer can be used interchangeably, as is the case for target, robber and evader. Unless the context demands otherwise, we use “robot” and “target” throughout the rest of this section.

**Cop Numbers**

Most graphs require more than one robot to capture the targets, see for example Figure 4.2. This concept is formalized by introducing the *cop number* $c(G)$ of a directed graph $G$, which is the minimal number of robots required to win, regardless of the initial positions of robots and targets. For example, the graph in Figure 4.1 has a cop number of 3.

Immediately the question arises how to compute $c(G)$ for a given graph. Another problem is to give sufficient properties for a graph $G$ so that $c(G)$ has a fixed value. In this thesis we are not concerned with answering these questions but rather how to find a winning strategy for $c(G)$ robots for a class of graphs that is as general as possible. Such a winning strategy clearly exists for $c(G)$ robots by definition.

### 4.3.2 Computational Considerations

For general graphs and if the robots have no information about the positions of the targets, finding the cop number is NP-complete ([13], Figure 1). Even when both robots and targets have complete information about the positions of all robots and targets, computing $c(G)$ in an undirected graph, which is a simpler problem than the one we are considering here, is EXPTIME-complete ([30], Theorem 3).

It was shown before that the synthesis of a GR[1] specification $\varphi$ can be done in polynomial time in the size of $\varphi$. However, we do not try here to solve an NP-complete problem by a polynomial time algorithm. When formulating a GR[1] specification $\varphi$ for a winning strategy, $\varphi$ must explicitly contain the system variables that control the robots, i.e. $cellID_j$ for all $j$. Therefore, the number of robots that is available to pursue the targets is already explicitly encoded in $\varphi$.

With such a specification, GR[1] synthesis merely decides whether a given number of robots is sufficient to win the game. However, this does not mean that the cops and robbers game is solved, which consists in *finding* $c(G)$. This must be
4.4. MOVING TARGET SEARCH STRATEGIES

done at the time of formulating the specification, because the system variables must be able to independently control at least \( c(G) \) robots.

From this discussion we conclude that the cops and robbers game cannot be solved by GR[1] synthesis: An NP-complete problem cannot be solved by the polynomial time algorithm for GR[1] synthesis. However, realizability of a specification \( \varphi \) of a winning strategy that explicitly includes the number of robots, can be decided by GR[1] synthesis. This is shown constructively in Section 5.2 by synthesizing controllers implementing a winning strategy.

Since GR[1] synthesis cannot be used to find \( c(G) \) and a GR[1] specification must have the number of robots encoded explicitly, we assume that the cop number \( c(G) \) is provided alongside a graph \( G \).

4.4 Moving Target Search Strategies

Instead of solving the cops and robbers game, we are interested in finding winning strategies for a given set of \( m \) robots \( \{R_0, R_1, \ldots, R_{m-1}\} \). Clearly, no such strategy exists if \( m < c(G) \). We first categorize the winning strategies given that a central authority \( C \) can directly control the moves of all robots, called globally coordinated winning strategies.

Thus the player \( C \) implements its strategy as an FDS \( M_\varphi \), which is specified by a GR[1] specification \( \varphi_\varphi \). The environment variables of \( M_\varphi \) are controlled by the player \( R \). However, no FDS implementation \( M_\varphi \) of the targets’ strategy is synthesized. The targets are simply assumed to always perform the worst possible moves for the robots.

The central authority \( C \) controls the system variables \( \text{cellID}_j \) for all \( j \). This is the standard formulation found in the literature. However, we go on to identify globally coordinated winning strategies that can also be specified by several GR[1] specifications that are synthesized into a set of FDS’s that together implement the globally coordinated strategy. This corresponds to reformulating the two-player cops and robbers game into a game where several robots \( C_1, C_2, \ldots, C_m \) play against the adverse targets \( R \). The winning strategies of the robots are based on local information in the graph only, and they must negotiate with each other in order to cooperate or share information. Therefore, such strategies are referred to as locally coordinated winning strategies.

Note that the requirement to provide sufficiently many robots means that the cop number must be known at specification time. It turns out, that the cop number is not even required to be included in the local specifications, but is used only to build the architecture for sufficiently many robots.
4.4. MOVING TARGET SEARCH STRATEGIES

4.4.1 The State of a Graph

We define the state of a graph to indicate to the robots whether the target can be said with certainty to not be on a particular edge or vertex. If the target cannot be on an edge, then the edge is contaminated, and the same notion is introduced for vertices and graphs in the following definitions:

Definition 11. An edge \( e \in E_G \) in a graph \( G = (V_G, E_G) \) can either be **contaminated** or **cleared**. Initially, all edges of the graph are contaminated. A contaminated edge \( e = (u, v) \) from \( u \) to \( v \) can be cleared by a robot either by placing a guard robot on \( u \) and sliding down the edge from \( u \) to \( v \) with another robot, or if all in-edges of \( u \) are already cleared by simply sliding down the edge from \( u \) to \( v \) with a robot, without requiring a guard on \( u \).

Definition 12. A vertex is **cleared** if all its in-edges and out-edges are cleared, **partially cleared** if all its in-edges are cleared and at least one of its out-edges is contaminated, **contaminated** if at least one of its in-edges and all of its out-edges are contaminated and no robot is guarding it, and **critical** otherwise. A contaminated vertex can be a **start vertex**, a concept which is introduced below.

The state of an edge \( e \in E_G \) is \( S_e(e) \in \{\text{ecl}, \text{eco}\} \), where ecl and eco correspond to “cleared” and “contaminated” respectively. Similarly, the state of a vertex \( v \in V_G \) is \( S_v(v) \in \{\text{vcl}, \text{vpc}, \text{vcr}, \text{vco}\} \) where vcl, vpc, vcr and vco correspond to “cleared”, “partially cleared”, “critical” and “contaminated” respectively. The **state of a graph** is the set of all states of its edges and vertices.

Definition 13. A graph is **cleared** if all its vertices are cleared, partially cleared, or are guarded by a robot, it is **contaminated** if all vertices and edges are contaminated, and it is **critical** if it is neither cleared nor contaminated.

4.4.2 Graph Clearing Sequences

A strategy defines a set of moves of the robots depending on the observations about the targets, which are obtained by querying the state of the graph, e.g. using sensors, see Section 4.1.3. We define strategies for the cops that assume that a central authority can control the movement of the robots. A list of robots \( R \) that are available to move is maintained as well as an auxiliary list of cops \( \mathcal{G} \) that keeps track of the guarding robots. Given that there are \( m \) robots \( \{R_0, R_1, \ldots, R_{m-1}\} \), we always have \( R \cup \mathcal{G} = \{R_0, R_1, \ldots, R_{m-1}\} \).

Definition 14. Given a graph \( G = (V_G, E_G) \) and sufficiently many cops in \( R \), a **clearing move** \( g \) is defined as a sequence of movements of robots that satisfies the following procedure:

(S1) if \( R \neq \emptyset \) then select \( R \in R \) and remove \( R \) from \( R \)

else select a guarding robot \( R \) from \( \mathcal{G} \)

Place \( R \) as a guard on a contaminated vertex \( v \in V_G \) and put \( R \) in \( \mathcal{G} \)

(S2) for each out-edge \( (v, u) \in E_G \) of \( v \) do

(S2.1) if \( R \neq \emptyset \) then select \( R' \in R \)
else select a guarding robot \( R' \) from \( \mathcal{G} \) and remove \( R' \) from \( \mathcal{G} \)
Place \( R' \) on \( v \)
(S2.2) 
Slide \( R' \) from \( u \) to \( v \) along \((v, u)\). \textit{This clears the edge} \((v, u)\)
(S3) 
while there is a partially cleared vertex \( p \in V_G \) do
(S3.1) 
if \( \mathcal{R} \neq \emptyset \) then select \( R'' \in \mathcal{R} \)
else select a guarding robot \( R'' \in \mathcal{G} \) and remove \( R'' \) from \( \mathcal{G} \)
Place \( R'' \) on \( p \)
(S3.2) 
for each contaminated out-edge \((p, q) \in E_G \) of \( v \) do
(S3.2.1) 
Slide \( R'' \) from \( p \) to \( q \) along \((p, q)\). \textit{This clears the edge} \((v, u)\)
(S4) 
if \( v \) is cleared then place \( R \) back into \( \mathcal{R} \)

This procedure is satisfied by a sequence of robot moves if the statements are completed sequentially, i.e. (S1) precedes (S2), (S2) precedes (S3), (S2.1) precedes (S2.2) and so on. Note that the dots indicate a hierarchy, so (S2) is completed only if (S2.1) and (S2.2) are. The contaminated vertex \( v \in V_G \) selected in (S1) is called the \textit{start vertex} of the clearing move.

The procedure presented in the above definition does not deterministically specify the moves of each robot at every time. The sources of nondeterminism are the following:

- There is no specific order according to which robot should be chosen from \( \mathcal{R} \) in (S1), (S2.1) or (S3.1).
- There is no specific order according to which the guarding robot should be chosen from \( \mathcal{G} \) in (S1), (S2.1) or (S3.1).
- Any contaminated vertex may be chosen in (S1).
- The partially cleared vertices may be chosen in any order in (S3).
- Robots that are in \( \mathcal{R} \) or not in \( \mathcal{G} \), and not sliding down an edge in (S2.2) or (S3.2.1) may move around on the graph freely.
- The order in which the edges are cleared on the sliding moves is not specified in (S2) and (S3.2).
- Only the order of necessary moves is specified, but there could be an arbitrary number of actions interleaved into the given sequence that do not violate the procedure (i.e. guards may not move).

While a clearing move does not completely specify all robot moves, it completely specifies the change in the state of the graph, depending only on the choice of the start vertex \( v \) in (S1) and the choice of the robots \( R, R' \) and \( R'' \) in (S1), (S2.1) and (S3.1) respectively. We adopt the convention that \( \mathcal{G} \) acts like a first in first out (FIFO) queue so that the robot that was first selected as guard will be the first robot to be selected to move away from the vertex it is guarding.

The effect of a sequence of clearing moves \( g_0 g_1 \ldots g_n \) on the state of the graph is therefore uniquely specified by the start vertices of the clearing moves.

\textbf{Definition 15.} A \textbf{graph clearing sequence (GCS)} for a graph \( G = (V_G, E_G) \) is a sequence of clearing moves \( \Gamma = g_0 g_1 \ldots g_l \) such that at the end of the clearing move \( g_l \) the graph is cleared.
4.4. MOVING TARGET SEARCH STRATEGIES

recontaminate = True
while recontaminate do
    recontaminate = False
    for each vertex \( v \in V_G \) do
        if no guarding robot on \( v \) and \( v \) cleared or contaminated then
            for each out-edge \( e = (v, u) \in E_G \) of \( v \) do
                contaminate \( e \)
                recontaminate = True

Table 4.2: Procedure ensuring recontamination is propagated appropriately.

A graph is initialized with all its edges and vertices being contaminated. Since we are only considering strongly connected graphs, there can be no partially cleared vertices initially if all edges are contaminated. The winning strategies then consist of sequences of clearing moves in which the robots move together to clear the graph.

**Theorem 1** ([52], Theorem 3.4). Given a contaminated graph \( G = (V_G, E_G) \) with cop number \( c(G) \) and at least \( c(G) \) cops in \( \mathcal{R} \), there exists a GCS \( \Gamma \) for \( G \).

When a guarding robot moves away from a critical vertex (i.e. a vertex with some cleared out-edges), there is the possibility of recontamination [52]. An edge \( e = (u, v) \in V_G \) is recontaminated if it has been previously cleared but the vertex \( u \) is no longer guarded, cleared or partially cleared. The recontamination of an edge can cause vertices to change their state, and thus further edges to change their state, possibly causing a cascade of recontaminations in the graph. The state of the graph can be updated using the procedure in Table 4.2.

This procedure is used in the implementation of the global storage, cf. Section 4.5.2.

It can be shown that no recontaminations are necessary to clear a contaminated graph \( G \) with no less than \( c(G) \) robots:

**Theorem 2** ([52], Theorem 2). Given a contaminated graph \( G = (V_G, E_G) \) with cop number \( c(G) \), at least \( c(G) \) robots in \( \mathcal{R} \) and a GCS \( \Gamma \) of \( G \), the set \( \mathcal{R} \) will always be nonempty at steps (S1), (S2.1) and (S3.1) of any clearing move in \( \Gamma \). That is, no guarding robot ever has to be removed and no recontaminations occur.

**Theorem 3.** Given a graph \( G = (V_G, E_G) \), \( c(G) \) robots in \( \mathcal{R} \), and a GCS \( \Gamma \) for \( G \), the graph is cleared by \( \Gamma \) independently of the initial state of \( G \).

**Proof sketch.** Denote \( G \) and \( \mathcal{R} \) at the beginning of \( \Gamma \) by \( \hat{G} \) and \( \hat{\mathcal{R}} \) respectively. Let \( \mathcal{S} \) be the set of the guarding robots at the beginning of \( \Gamma \) (some robots must be guarding, otherwise the graph must be cleared and Theorem 1 yields the result).

\[ {68} \]

\[ {68} \] Of course, \( c(G) = c(\hat{G}) \).
4.4. MOVING TARGET SEARCH STRATEGIES

When executing $\Gamma$ on $\hat{G}$, at steps (S1), (S2.1) and (S3.1), the robots in $\hat{\mathcal{G}}$ will be chosen before the robots that have been removed previously from $\hat{\mathcal{R}}$ in step (S1). On using the guarding robots from $\hat{\mathcal{G}}$, recontamination may happen. By Theorem 2, during the execution of $\Gamma$, no guarding robots other than those in $\hat{\mathcal{G}}$ are required in steps (S1), (S2.1) or (S3.1). Therefore, recontamination never affects edges that have been cleared during the execution of $\Gamma$ in steps (S2.2) and (S3.2.1) and so $\Gamma$ clears $G$ independently of the initial state of $G$.

**Corollary 1.** Given a contaminated graph $G = (V_G, E_G)$, $c(G)$ robots in $\mathcal{R}$, a GCS $\Gamma$ for $G$, and a finite sequence of clearing moves $\Gamma' = g_0g_1\ldots g_n$, the concatenation $\Gamma'\Gamma$ is also a GCS for $G$.

**Proof.** Let $\hat{G}$ and $\hat{\mathcal{R}}$ be $G$ and $\mathcal{R}$ at the end of $\Gamma'$ respectively. At the end of $\Gamma'$, $\hat{G}$ is either contaminated, critical or cleared, and $|\hat{\mathcal{R}}| \leq c(G)$. The result follows from Theorem 3.

Corollary 1 states that the set of graph clearing sequences is closed under prefixing with clearing moves. When developing a globally coordinated winning strategy for the robots, it is therefore sufficient to ensure that the sequence of robot moves it entails eventually ends in a GCS. As mentioned above, it suffices to specify the order of starting vertices of the clearing moves. Thus, a winning strategy can be defined by a function from the state of the graph to the next starting vertex.

Since we are ultimately interested in finding a locally coordinated winning strategy, the choice of the next start vertex should not depend on the entire state of the graph, because in a real implementation this information is not available to the robots without a central authority to coordinate them. Therefore, let $V^i_G$ be the set of vertices for which a robot can observe the state before the $i^{th}$ clearing move. It is defined to be $V_G$ for $i = 0$ and for $i \leq 0$ by

\[
v \in V^i_G \Leftrightarrow \exists e = (p, q) \in E_G, S_e(e) = \text{ccl} \land (p = v \lor q = v \lor \exists f = (r, s) \in E_G, (r = v \land (p = s \lor q = s) \lor s = v \land (p = r \lor q = r))).
\]

That is, before the $i^{th}$ clearing move $g_i$ in $\Gamma$, a robot can observe the state of a vertex $v \in V_G$ iff $v$ is the destination or source of an edge $f$ that goes to a vertex $u \in V_G$ which itself is the destination or source of a cleared edge $e$. Hence, the robots have a limited visibility radius of the graph state (while still only being able to sense a target in the same cell).

**Definition 16.** Let $E^i_G$ be the set of cleared edges before the $i^{th}$ clearing move $g_i$ in $\Gamma$ and $\langle E_G^i \rangle$ the graph induced by the edges in $E^i_G$. Let $C^i_G \subseteq V^i_G$ be the set of critical or contaminated vertices in $V^i_G$ that have at least one contaminated in-edge.

**Definition 17.** Let $\mu_i$ be the minimum number of contaminated in-edges of any vertex $v$ in $C^i_G$ if $C^i_G$ is nonempty, and let $\mu_i = \infty$ otherwise. A vertex $v \in V^i_G$ is a $\mu_i$-clear-candidate if it has exactly $\mu_i$ contaminated in-edges before the $i^{th}$ clearing move $g_i$ in $\Gamma$. 

69
### 4.4. MOVING TARGET SEARCH STRATEGIES

**Definition 18.** Let $\nu_i$ be the maximum number of contaminated out-edges to $\mu_i$-clear-candidates of any vertex in $V_i^G$ if $\mu_i < \infty$, and let $\nu_i = -\infty$ otherwise. A vertex is a $\nu_i$-start-candidate if it has exactly $\nu_i$ out-edges to $\mu_i$-clear-candidate vertices before the $i$th clearing move $g_i$ in $\Gamma$.

The start vertex $v^i \in V_G$ of the $i$th clearing move $g_i$ in $\Gamma$ is selected according to the following heuristics, denoted by $\mathcal{H}$. The six criteria are listed in order of decreasing priority:

1. **(H1)** The vertex $v^i$ is contaminated.
2. **(H2)** The graph $\langle E^{i+1}_G \rangle$ must be connected (but not necessarily strongly connected).
3. **(H3)** If it exists, select a $\nu_i$-start-candidate vertex $v^i \in V_i^G$ which is not a $\mu_i$-clear-candidate.
4. **(H4)** Otherwise, if it exists, select a $\nu_i$-start-candidate vertex $v^i \in V_i^G$ which is also a $\mu_i$-clear-candidate.
5. **(H5)** Select the vertex with the minimum number of in-edges.
6. **(H6)** Select the vertex with the maximum number of out-edges.

Note that even after (H6) there still might be several possibilities of choosing $v^i$ in which case the start vertex is chosen nondeterministically.

#### 4.4.3 Evaluation of the Strategy

With these heuristics, the $i$th start vertex is selected from the contaminated vertices in $V_i^G$. This is a local decision in order to solve the global problem of finding the best starting vertex. This requires to make assumptions on the graph that permit the local heuristics to always lead to a global solution, i.e., choosing the start vertices according to the local heuristics successfully clears the graph $G$ with exactly $c(G)$ robots.

When choosing the start vertices according to the above heuristics $\mathcal{H}$, it is hoped that a GCS results. However, this is not true for all graphs. One possible counterexample is the graph in Figure 4.3. This is a planar graph with just five vertices, but still the heuristics do not necessarily produce a GCS. A GCS would arise from choosing to guard $v_1$ and $v_4$ (in any order). However, the heuristics may choose from any of the following sequences of start vertices: $v_2v_0v_3$ or $v_2v_0v_4$ or $v_4v_1$. However, the former two require four robots, while the cop number of the graph is just three.

The heuristics would be easy to fix for this graph by adding another criterion

7. **(H7)** In a tie, sum up the contaminated out-edges of the destinations of each vertex and choose the vertex with the maximum such number.

---

*The same principle is used when optimizing convex functions using local decisions, since the global assumption of convexity means that any local optimum is a global optimum.*
Figure 4.3: An example graph with five vertices for which the heuristics fail. Its cop number is three, but with the heuristics \( H \), four cops might be required (although not in all cases.)

| Vertices | Samples | Counterexamples |
|----------|---------|-----------------|
| 5        | 1000    | 1               |
| 6        | 1000    | 4               |
| 7        | 1000    | 11              |
| 8        | 1000    | 16              |
| 9        | 1000    | 23              |
| 10       | 1000    | 42              |

Table 4.3: Counterexamples to the heuristics \( H \). Random graphs generated according to Section 4.1.2.

Even if this would give rise to a GCS for this particular graph, it does not however solve the general problem: We try to find a simple solution to a provably hard problem, i.e. solving a global problem using only local information.

In general, the larger the graphs get, the more likely it is to encounter graphs that cannot be cleared using the heuristics with just \( c(G) \) robots. Table 4.3 presents the results of simulations for a number of small graphs. For various counts of vertices, random graphs have been generated and the number of instances in which \( H \) does not result in a GCS are listed as counterexamples. The larger the graphs, the more likely it is for \( H \) to not yield a GCS. In the worst case of the explored examples in Table 4.3 even \( c(G) + 4 \) robots are required.

Reformulation as Optimization Problem

Finding the minimum number of robots required to clear a graph can be formulated as an optimization problem. In order to evaluate the cop number of a graph with \( n \) vertices, we consider an objective function \( c_{n}(G, \delta) : S_{n} \times V_{G}^{+} \rightarrow \mathbb{N} \) from graphs \( G \in S_{n} \) and sequences of starting vertices \( \delta \in V_{G}^{+} \) to the number of robots required to clear the graph when placing them on the start vertices \( \delta \) in (S1) of the clearing moves.\(^7\) The function \( c_{n} \) is potentially a very complicated function, and, as the results presented in Section 4.3.2 show, requires exponential time to compute in the number of vertices \( n \).

\[^7\]Recall from definition 4.1.1 that \( S_{n} \) is the set of strongly connected graphs with \( n \) vertices and from Definition 4 that \( V_{G}^{+} \) is the set of nonempty sequences over \( V_{G} \).
4.5. **LOCALLY COORDINATED STRATEGIES**

We now minimize \( c_n \) over the sequences of starting vertices for some given graph \( G \in \mathcal{G}_n \). When all sequences of starting vertices are feasible, then the minimum will be equal to the cop number \( c(G) \) by definition:

\[
P_1 : \quad c(G) \triangleq \min_{\delta \in V_G^+} c_n(G, \delta).
\]

Encoding the graph \( G \) as an adjacency matrix \( \text{Adj}(G) \) and \( \delta \) as a vector may yield a combinatorial optimization problem. Solving this is of course NP-complete, corresponding to the complexity of computing the cop number.

Now, the heuristics can be interpreted as constraints on the set of feasible sequences of starting vertices. Here we denote a heuristics simply by a propositional function \( h(G, \delta) : \mathcal{G}_n \times V_G^+ \to \mathbb{B} \) that is true iff the \( \delta \) satisfies the heuristics \( h \) in the graph \( G \). Finding the number of robots under a heuristics \( h \) can thus be reformulated as the optimization problem

\[
P_2 : \quad \min_{\delta \in V_G^+} c_n(G, \delta)
\quad \text{s.t.} \quad h(G, \delta) = \text{True}.
\]

Showing whether a heuristics \( h \) is optimal, in the sense that it always chooses the best start vertex so that only \( c(G) \) cops are required to clear the graph is equivalent to proving that for all graphs \( G \), the solutions to the optimization problems \( P_1 \) and \( P_2 \) yield the same value. This can be written as:

\[
h \text{ optimal } \iff \forall G \in \mathcal{G}_n . \min_{\delta \in V_G^+} c_n(G, \delta) = \min_{\delta \in V_G^+} c_n(G, \delta) \quad \text{s.t.} \quad h(G, \delta) = \text{True} \quad (4.4)
\]

Table 4.3 clearly shows that the heuristics \( H \) we developed is not optimal. However, even if we could develop an optimal strategy, verifying this using (4.4) would be a nontrivial task.

### 4.5 Locally Coordinated Strategies

So far we assumed that a central coordinating authority exists that can directly control the robots. However, we are interested in developing a locally coordinated winning strategy. Several questions must be answered: How to adapt the globally coordinated winning strategy to a locally coordinated one? How is authority negotiated between the robots, i.e. how do the robots coordinate? Where is the information about the state of the edges and vertices of the graph stored and how is it accessed or observed by sensors? This section is dedicated to answering these questions, and a complete specification is presented in Section 5.2.

#### 4.5.1 Negotiations

Given a graph \( G \) and its cop number \( c(G) \), we would like to find \( c(G) \) strategies, one for each robot, that together implement a winning strategy for \( G \). No central authority is used, and so the robots must somehow cooperate in order to
Eventually find the targets. The required communication protocol has already been presented in Section 3.2.2.

Coordinating their movements on the graph restricts the authority each individual robot has. For example, a guarding robot may not move from a vertex $v$ while another robot is engaged in sliding down the out-edges of $v$. This requires the robots to be able to negotiate for authority over each other, so that one robot can, at least temporarily, control the other.

Instead of implementing sophisticated negotiation algorithms in the specification, we fix the structure of authority. Thus, each robot has a master and a slave, resulting in a circular structure of authority. Indeed this is necessary when recontamination occurs and the robot that was the first guarding robot in the GCS is told to move by its master.

This results in symmetrical specifications of the robots, where each robot provides the same services to its master and may expect the same services from its slave. We therefore develop only one local specification $\varphi$ and merely change the names of the variables specific to a robot. In practice, $\varphi$ is synthesized once and the architecture ensures that the communication channels are set up correctly, see Section 3.2.1.

Thus, negotiation is decided at specification time and no longer has to be taken account in the specification. Communication is then merely used to transfer the commands from masters to slaves. The data, i.e. the state of the graph, can directly be obtained from the global storage discussed next.

### 4.5.2 Global Storage

Our implementation of the locally coordinated strategy for the cops and robbers game does not require the robots to have knowledge about the entire state of the graph. Instead, the robots are only provided with the information that is immediately required to form a local decision about the next move. This is achieved by storing the entire state of the graph in a data structure that can be accessed by all robots, but only provides local information to each robot.

Keeping the state outside of the searchers is current practice in SAR and is not a new idea. For example, the United Nations INSARAG marking system defines a set of shapes and letter used to mark a building’s status to aid the SAR operations. Similar techniques could be implemented for the SAR robots, but we only consider the abstraction to the graph state introduced above.

In order to formally state a specification of a distributed solution, we must define how to access the global state. A robot will only ever have access to the state of the vertex it is currently on. We thus include the system variable $\text{cst}_j \in \{\text{cl}, \text{pc}, \text{cr}, \text{sv}\}$ in the system variables of robot $R_j$, indicating the current

---

8INSARAG Guidelines and Methodology, United Nations Office for the Coordination of Humanitarian Affairs, Jan 2011. [http://ochanet.unocha.org/p/Documents/INSARAG%20Guidelines%202011-Latest.pdf](http://ochanet.unocha.org/p/Documents/INSARAG%20Guidelines%202011-Latest.pdf) accessed 03. Nov 2011
4.5. **LOCALLY COORDINATED STRATEGIES**

state of the vertex the robot is on. From the point of view of the robot this acts like a sensor that reads the relevant aspect of the environment.

The variable $\text{cst}_j$ takes the value $\text{sv}$ if $R_j$ is on a start vertex and $(S1)$ is being executed. In $(S1)$, $R_j$ is placed on a contaminated vertex $v$, for which $S_v(v) = vco$. However, $R_j$ may only choose a start vertex according to the heuristics $\mathcal{H}$. Whether a contaminated vertex is a start vertex is indicated to the robot by the global storage. Thus, if $S_v(v) = vco$, but $v$ is not a start vertex, then $\text{cst}_j = \text{cr}$ from the point of view of the robot $R_j$ if it is on $v$. However, if $v$ is a starting vertex and $R_j$ is on $v$, then $\text{cst}_j = \text{sv}$. Otherwise, $\text{cst}_j$ takes the values cl, pc and cr if $S_v(v) = vcl$, $S_v(v) = vpc$ and $S_v(v) = vcr$ respectively. This will later be used in the specification of the controllers in Section 5.2.2.

In an implementation, it has to be ensured that the robot indeed gets the current state of the vertex on which it is. This requires in particular that the state of the graph is updated appropriately. Initially, all edges of the graph are contaminated. The state of the vertices only changes with the state of the edges to which they are connected, and the state of the edges changes depending on the movement of the robots on the graph. Thus, the global state of the graph changes only when a robot slides down an edge in steps (S2.2) or (S3.2.1) or the graph is recontaminated. In the latter case the procedure in Table 4.2 is executed.

Keeping the state of the graph outside the robots has several advantages: The robots do not need to communicate the local information they have to each other, reducing the need for system and environment variables for communication. Furthermore, the state space of the robots decreases significantly by only querying the immediately relevant local information of the graph. Note that the state of a graph with $n = |V_G|$ vertices is uniquely defined by the states of its edges, giving rise to $O(2^{n^2})$ configurations in dense graphs.

4.5.3 **Multi-Stage Specification**

The local specifications must ensure that all robots together execute a sequence of movements that satisfies a GCS, i.e. a sequence of clearing moves. Thus a robot may be in one of four modes corresponding to the possible movements necessary to implement a clearing move. The robot $R_j$ with index $j$ therefore has a system variable $M_j \in \{\text{cs}, \text{gu}, \text{sp}, \text{sl}\}$ keeping track of the mode of the robot. If $M_j = \text{cs}$, the robot is searching for a (contaminated) starting vertex in (S1). If $M_j = \text{gu}$, the robot is guarding a vertex in (S1). If $M_j = \text{sp}$, the robot is searching a partially cleared vertex in (S3). And if $M_j = \text{sl}$, the robot is sliding down the out-edges of a vertex in (S2.2) or (S3.2.1).

All robots except one are initialized to guard some arbitrary vertex in the graph, i.e. they are in mode $M = \text{gu}$. The remaining robot is initialized on some arbitrary vertex but is searching for a starting vertex, i.e. it is in mode $M = \text{cs}$. For reference, a summary of the environment and system variables of the robots is given in Table 5.8 and Table 5.9 respectively.
4.5. LOCALLY COORDINATED STRATEGIES

Searching for Start Vertices: \( M = cs \)

A robot \( R_j \) in mode \( M_j = cs \) searches for the contaminated vertex that is to be the next start vertex in the sequence of clearing moves \( \Gamma \) that constitutes a GCS for the graph \( G \). The robot searches all vertices, querying their states until a start vertex is found, indicated by \( \text{cst}_j = \text{sv} \). This search is done in a circular fashion as outlined in Section 4.2.2. Using a circular search, the robot knows when all vertices have been visited and no contaminated vertex is in the graph. In this case, the graph is known to be cleared already.

If a contaminated vertex \( v \) is found that matches the six criteria in the heuristics \( \mathcal{H} \), it is selected as start vertex. Consequently, the robot enters guard mode \( M_j = \text{gu} \) and stays at \( v \) until called by its master to move away. On entering guard mode, the robot’s slave, with index \( j' \), is instructed to also move to \( v \) and start sliding down the out-edges of \( v \) to clear them. Thus the slave is forced to enter the search for a partially cleared vertex with \( M_{j'} = \text{sp} \).

Guarding: \( M = \text{gu} \)

A robot \( R_j \) in mode \( M_j = \text{gu} \) is guarding the vertex \( v \) on which it is currently positioned. Note that \( v \) must necessarily be a start vertex. \( R_j \) thus stays at \( v \) until instructed by its master to move away. The only way the master \( R_{j''} \) can force robot \( R_j \) to leave \( v \), is to be guarding a vertex \( w \) itself and to then instruct robot \( R_j \) to enter the clearing mode \( M_j = \text{sl} \) and clear all out-edges of the vertex \( w \) guarded by the master.

Searching for Partially Cleared Vertices: \( M = \text{sp} \)

This mode is very similar to searching for start vertices. However the difference is that the robot \( R_j \) reacts to finding a partially cleared vertex \( v \) by directly entering the clearing mode \( M_j = \text{sl} \). A partially cleared vertex does not need to be guarded, and thus in this case the robot that finds such a vertex does not need to instruct its slave to help clearing the out-edges of \( v \).

Clearing: \( M = \text{sl} \)

In the clearing mode \( M_j = \text{sl} \), robot \( R_j \) is using sliding moves to clear the out-edges of a partially cleared or guarded vertex \( v_i \). The vertex \( v_i \) is stored in the robot’s system variable \( \text{store}_j = i \), which is set on entering the clearing mode. The out-edges of \( v \) are cleared one at a time, which is specified in the guarantees of the robot’s specification. Since the order of edges to be cleared by sliding is arbitrary in the definition of a clearing move (see the discussion of sources of nondeterminism below the definition), it is perfectly admissible to fix the order at specification time.

To keep track of which out-edges of a vertex have already been cleared, the robot keeps a system variable \( c_{ij} \), which is a counter that is initialized to zero when the clearing mode is entered, in addition to setting \( \text{store}_j \) to the index \( i \) of
4.5. **LOCALLY COORDINATED STRATEGIES**

the vertex $v_i$. Whenever the robot slides down an out-edge of $v_i$, $c_j$ is increased by one. This is guaranteed by properties similar to the following:

$$\bigwedge_{i \in [0,n)} \bigwedge_{\gamma \in [0,\bar{\gamma}_i)} \Box(\text{store}_j = i \land c_j = \gamma \land X^j_i \rightarrow \bigcirc(X^j_{w^i_\gamma} \land c_j = \gamma + 1)),$$

where $\bar{\gamma}_i$ is the number of out-edges of the vertex $i$ and $w^i_\gamma$ is the (unique) vertex such that there is an edge $e$ from $i$ to $w^i_\gamma$ and $e$ is the $\gamma$th out-edge to be cleared by the robot.

After clearing an edge, the robot has to return to the stored vertex in $\text{store}_j$ again, which is guaranteed by properties similar to

$$\Box(\bigvee_{i \in [0,n)} (\text{store}_j = i \land \bigvee_{\gamma \in [0,\bar{\gamma}_i)} (c_j = \gamma + 1 \land X^j_{w^i_\gamma}) \rightarrow \Diamond \bigvee_{i \in [0,n)} (\text{store}_j = i \land X^j_i)),$$

The full specifications that follow the ideas developed in this section are given in Section 5.2.
Chapter 5

Implementations

In this chapter, we present two implementations of controllers for SAR robots, one for stationary target search in an USAR application, and the other for moving target search in a WiSAR application. In the stationary target search, a central authority, called the “allocator” is responsible for dispatching the robots to the vertices in the graph where a target needs to be rescued. In the moving target search, a distributed solution is adopted, where the authority structure among the robots is already negotiated in advance, cf. Section 4.5.1.

5.1 SAR of Stationary Target

In USAR, the targets are mostly considered stationary. We develop specifications to build controllers for a team of robots that is managed by a central allocator. The goal is to find all targets in the affected area and rescue them. A robot that is at the location of a target may engage with it by performing a rescue task like providing medical support. In this scenario, we will consider a target to be rescued if $\alpha = 2$ robots have engaged with it for a sufficient amount of time.

The allocator is responsible for gathering information about the targets’ status and location, and coordinating the action of the robots by dispatching them to the location of a target. If a robot is not dispatched, it may move around freely. The topology is represented as a graph, which is explained in Section 4.1.1. It is optimistically assumed that the entire graph is known to the robots at synthesis time and that it does not change during the SAR task.

When the allocator dispatches a robot to a vertex, it merely asserts the index of a target’s vertex on a transmission variable to the respective robot. The robot is then expected to move to this vertex on its own account, so the allocator does not need to know the topology. The allocator knows about a target’s needs by receiving a message, or a “flag” associated with the location of the target. Targets will be assumed to be stationary since an injured or trapped person will not move.
5.1. SAR OF STATIONARY TARGET

Communication between the allocator and the robots is assumed to be layered on top of some reliable wireless protocol. Delivery of data is guaranteed eventually in finite time, messages are not corrupted, no reordering takes place, no messages get lost — comparable to a data-layer protocol. Thus the communication is based on the four-phase handshake protocol developed in Section 3.2.3.

5.1.1 Architecture

We synthesize the controllers for a set of robots and the central allocator. The asynchronous composition of the reactive systems resulting from the synthesis must satisfy the global specification of rescuing each flagged target, see Section 4.2.1.

In order to be able to simulate the reactive systems, it is necessary to close them as explained in Section 2.2.1. This is done by also synthesizing a reactive system for each vertex or “cell” in the topology. These reactive systems control the flags according to the environment assumptions of the allocator, which gathers the information about the targets. The complete architecture is shown in Figure 5.1 and the individual parts are explained below.

Allocator and Queues

At the heart of the architecture is the allocator, denoted by $AMM$. It is connected to the robots, $\{R_0, R_1, \ldots, R_{m-1}\}$. The cells are also included in the figure to show how the closed-loop system would be controlled. They are shown as FDS’s $\{C_0, C_1, \ldots, C_{n-1}\}$.

The allocator is informed about whether a target needs to be rescued by the boolean flags $f_0, f_1, \ldots, f_{n-1}$, each associated with one cell. Similarly, the allocator knows about which robots are ready by receiving the boolean variables $r_0, r_1, \ldots, r_{m-1}$, each associated with one robot. If a robot is ready and a vertex has been flagged, the allocator must dispatch a robot to the vertex that has been flagged for the longest period of time.

If there are $n$ vertices and $m$ robots, the allocator would need at least $2^n + m$ states to account for the $n$ flags and $m$ ready signals of the robots. However, in order to reduce the allocator’s state space, two FIFO queues $FIFOf$ and $FIFOr$ are implemented as separate components that maintain which target and robot has been waiting for the longest period of time respectively. Thus the environment variables of $FIFOf$ are the flags $f_0, f_1, \ldots, f_{n-1}$ and the environment variables of $FIFOr$ are the ready signals $r_0, r_1, \ldots, r_{m-1}$.

The allocator receives a flag-index $F \in [\epsilon, n)$ from $FIFOf$, and a robot-index $R \in [\epsilon, m)$ from $FIFOr$ indicating which robot may be dispatched to which cell. If $FIFOf$ holds $f_i$ at its head, then it offers $F = i$ to the allocator, and symmetrically if $FIFOr$ holds $r_j$ at its head, then it offers $R = j$. Moreover, the allocator may request the next flag index from $FIFOf$ by sending a dequeue signal $DEQf$, which is acknowledged by $DEQf\_ACK$. The queue $FIFOr$ does
not have such a dequeuing capability, since this is not required in the specification.

The allocator can dispatch robots by setting the dispatch variable \( D \in [\epsilon, m] \) to the value in \( F \). This indicates to the robot which is currently selected by \( R \) that it has to go to the vertex in \( D \). Internally, \( AMM \) maintains a counter variable \( c \in [0, \bar{\alpha}] \) of the number of robots that have already been dispatched to the cell with index \( F \). If possible, \( \bar{\alpha} = 2 \) robots will be dispatched to a cell, but never more, because \( \bar{\alpha} \) is the number of robots that are required to clear the flag.

We do not provide specifications for the queues since they are not synthesized in the way explained above. With \( n \) inputs a FIFO FDS would have \( O(2^n n!) \) states, making synthesis futile even for modest \( n \). This number results from the \( O(n!) \) configurations the internal store of the queue can be in (corresponding to the system variables \( Y \)) and the \( O(2^n) \) configurations the boolean inputs can be in (corresponding to the environment variables \( X \)). Hence, we consider FIFO’s as standard components and use a Python implementation for simulation.

Robots
For stationary target search, the number of robots required is only as high as the number of robots required to rescue a target, i.e. \( \bar{\alpha} \). However, we may include an arbitrary number of robots in the architecture to complete the rescue operation more effectively. For example, with \( \bar{\alpha} = 2 \), if there are two targets and four robots, the robots can be dispatched in teams of two and rescue both targets at the same time. We synthesize one controller that is used on all robots and simply rename the variables with indices to indicate which variable belongs to a particular robot.

A robot \( R_j \) maintains its own position on the topology both in \( cellID_j \in [\epsilon, n] \) and in the boolean variables \( X^j_0, X^j_1, \ldots, X^j_{n-1} \). As explained above, the two representations are merely introduced for notational convenience in the specification. Another variable, \( store_j \in [\epsilon, n] \), stores the destination cell if the robot has been dispatched.

The robot \( R_j \) knows that it is dispatched from the dispatch signal \( D \) sent by \( AMM \), if the associated robot index \( R \) from \( FIFO_r \) satisfies \( R = j \), indicating that \( AMM \) sends \( D \) to \( R_j \). However, the robot is not interested in whether any other robot is dispatched or not. Therefore, in order to decrease the state space of the robot, auxiliary environment variables \( d_0, d_1, \ldots, d_{m-1} \) over the domain \( [\epsilon, n] \) are introduced, one for each robot. Robot \( j \) receives \( d_j = i \) from the allocator if it is dispatched to vertex \( v_i \). On reaching the vertex \( v_i \), the robot checks its boolean sensor signal \( flag_j \) whether there is still a target on this vertex. The environment variable \( flag_j \) is true iff there is a target on the vertex the robot is currently on. If a target is detected at the vertex to which the robot is dispatched (recognized by \( cellID_j = store_j \land flag_j = \text{True} \)), the robot asserts a boolean engage signal \( e_j \) to indicate to the allocator that it has successfully reached the target’s vertex.
5.1 SAR OF STATIONARY TARGET

Figure 5.1: Allocator based SAR architecture. Implicitly, $i \in [0, n)$ and $j \in [0, m)$. The outgoing arrows showing the system variables $e_j$ of the robots are omitted for a clearer layout. Also, internal system variables such as $c$ and $store_j$ are not shown.
Connectors

The communication between the individual components in the architecture is shown by the arrows between the individual components. Some environment variables of the robots and cells are functions of the system variables of their peers. For example, the environment variable $E_i$ of cell $C_i$ is a function of the variables $cellID_j$ and $e_j$ for $j \in [0, m)$. One way to incorporate their relationship into the design would be to have $E_i$ as a system variable and $cellID_j$ and $e_j$ for $j \in [0, m)$ as environment variables in $C_i$. Then the guarantee

$$\Box (E_i = \sum_j (e_j \land X^j_i))$$

can be added to the specification of $C_i$. Note that the summation interprets “True” as “1” and “False” as “0”. While this would produce the desired result, it introduces many environment variables into the design of the cells. If they cannot be suitably restricted by assumptions in the specification, this will lead to a considerable increase in the number of states in each cell’s FDS.

To overcome this, these functions are implemented externally to the FDS’s in the form of connectors. A connector is an invariant proposition of the global state space. It is of the form $\vartheta = \pi(\vartheta_0, \vartheta_1, \vartheta_2, \ldots)$ where $\vartheta$ is an environment variable of an FDS $\mathcal{M}$, called a connector variable, $\vartheta_0, \vartheta_1, \vartheta_2, \ldots$ are system variables of peers of $\mathcal{M}$ and $\pi$ is some proposition over $\vartheta_0, \vartheta_1, \vartheta_2, \ldots$ with the additional condition that $\vartheta$ is not a system variable of any component in the architecture and $\vartheta$ is an environment variable only of $\mathcal{M}$. Thus, a connector introduces a new variable to the global state variables.

When proving properties over the composition of the specifications, these invariants can simply be assumed to hold at all times. In a simulation, these invariants are ensured by calculating the values of a connector variable $\vartheta$ whenever it is accessed by an FDS.

In our implementation of stationary target search, there are three types of connectors, shown as rectangles in Figure 5.1. There is a connector for the engage signal $E_i$ of a cell $C_i$ that indicates how many robots are engaged on vertex $i$. This is expressed by

$$E_i = \sum_j (e_j \land X^j_i). \quad (5.1)$$

Another connector is for the flag signal $flag_j$ of a robot $R_j$ that indicates whether the vertex $i = cellID_j$ the robot is currently on contains a target, i.e. $f_i = True$. This expressed by

$$flag_j = \bigvee_i (X^j_i \land f_i). \quad (5.2)$$

The third type of connectors is for the dispatch signal $d_j$ of a robot $R_j$. Robot $R_j$ is dispatched to the vertex $i$ iff $d = i$ and $r = j$, i.e. if the robot that is offered to the allocator is $R_j$ and the allocator decides to dispatch a robot to the vertex $i$. This is expressed by

$$d_j = \begin{cases} 
D, & \text{if } R = j \\
\epsilon, & \text{otherwise}
\end{cases}. \quad (5.3)$$

Note that the connectors have access to the global state, i.e. they are not restricted by the position a robot is in. However, the expressions in 5.1–5.3
5.1. SAR OF STATIONARY TARGET

| Name      | Domain | Description                  |
|-----------|--------|------------------------------|
| R         | $[\epsilon, m)$ | Indicates a robot that is ready |
| F         | $[\epsilon, n)$ | Indicates a vertex that is flagged |
| DEQf_ACK  | B      | Dequeue acknowledge signal from FIFO |

Table 5.1: Environment variables of the allocator, AMM.

| Name | Domain | Description                  |
|------|--------|------------------------------|
| DEQf | B      | Dequeue signal to FIFO       |
| c    | $[0, \bar{\alpha}]$ | Counts how many robots have been dispatched |
| D    | $[\epsilon, m)$ | Dispatch signal to robots     |

Table 5.2: System variables of the allocator, AMM.

clearly indicate that only the local state is used to calculate the connector variables. By local state we mean that a cell can only use the values of robots on the corresponding vertex and a robot can only use the values of the cell on which it is currently.

5.1.2 Global Specifications

The composition of all components in Figure 5.1 has to satisfy the global specification for stationary target search discussed in Section 4.2.1 in a form that corresponds to the problem setting considered here. A target on vertex $i$ is indicated by the flag $f_i$ being asserted, and the goal of the allocator is to get $\bar{\alpha} = 2$ robots to arrive at the vertex $i$. This can be expressed analogously to (4.1) by

$$\bigwedge_{i \in [0,n)} \square (f_i \rightarrow \Diamond \bigvee_{j_1 \in [0,m)} \{j_1\} (X_{j_1}^{i} \land X_{j_2}^{i})).$$

We now state the specifications of each component so that the asynchronous composition fulfills this global specification. This can be verified for example by using the Feedback Interconnection Refinement Rule in Proposition 1, but here we only note that the simulation yields the desired result.

For compacter specifications, define

- $\text{stay}(\pi) \triangleq \pi \land \Box \pi$,
- $\text{raise}(\pi) \triangleq \lnot \pi \land \Box \pi$,
- $\text{clear}(\pi) \triangleq \pi \land \Box \lnot \pi$,

for any propositional formula $\pi$.

5.1.3 Allocator Specification

The state variables of the allocator are shown in tables Table 5.1 and Table 5.2. The following specifications are for the FDS AMM in Figure 5.1.
Assumptions

Initially, no robot is ready, so \texttt{FIFOr} offers no robot and \( R = \epsilon \). Moreover, \texttt{FIFOf} doesn’t have a dequeue request to acknowledge, so \( \text{DEQf}_{\text{ACK}} \) is low:

\[
\varphi^c_{A,1} = (R = \epsilon) \land \neg\text{DEQf}_{\text{ACK}}.
\]

\texttt{FIFOr} keeps track of the robots that are ready, and so, if available, presents a robot to the allocator that can be dispatched. This robot will not change unless the formerly presented robot has been dispatched:

\[
\varphi^c_{A,2} = \bigwedge_{j \in [0,m)} (\text{clear}(R = j) \rightarrow D \neq \epsilon).
\]

Once a vertex is flagged, it is not unflagged unless a robot is dispatched to this cell, i.e. no target can rescue itself. More precisely, the flag offered by \texttt{FIFOf} may only change if a robot is dispatched or a dequeue signal is sent to the queue:

\[
\varphi^c_{A,3} = \bigwedge_{i \in [0,n)} (\text{clear}(F = i) \rightarrow \text{DEQf} \lor D \neq \epsilon).
\]

The dequeue acknowledgement stays high as long as the dequeue request stays high:

\[
\varphi^c_{A,4} = \Box(\text{DEQf}_{\text{ACK}}) \lor \neg\text{DEQf}.
\]

Guarantees

Initially, the dequeue signal is low, the counter is zero and no robot is dispatched:

\[
\varphi^s_{A,1} = \neg\text{DEQf} \land c = 0 \land D = \epsilon.
\]

Once the allocator has dispatched a robot, it monitors the ready signal it receives from \texttt{FIFOr}. As long as this does not change, it does not clear the dispatch signal:

\[
\varphi^s_{A,2} = \bigwedge_{j \in [0,m)} \bigwedge_{i \in [0,n)} (D = i \land \text{stay}(R = j) \rightarrow \Box(D = i)).
\]

However, if \texttt{FIFOr} presents a different robot to the allocator, this means that the robot to which the dispatch signal has been sent is no longer ready and has successfully reacted to the dispatch request. Thus dispatch is cleared:

\[
\varphi^s_{A,3} = \bigwedge_{j \in [0,m)} \bigwedge_{i \in [0,n)} (D = i \land \text{clear}(R = j) \rightarrow \Box(D = \epsilon)).
\]

If the allocator gets offered a flag \( F = i \) from \texttt{FIFOf} and a robot is offered by \texttt{FIFOr}, it may dispatch a robot to vertex \( i \) by asserting \( D = i \). In this case the counter \( c \) must be increased by one:

\[
\varphi^s_{A,4} = \bigwedge_{\gamma < \alpha} \bigwedge_{i \in [0,m)} (F = i \land D = \epsilon \land c = \gamma \land R \neq \epsilon \rightarrow \Box(D = i \land c = \gamma + 1)).
\]
5.1. SAR OF STATIONARY TARGET

The dispatch signal may only be asserted if the counter \( c \) is not saturated, some robot is ready and a cell is flagged:

\[
\varphi^*_A,5 = \bigwedge_{i \in [0,m]} \square(D = \epsilon \land \bigcirc(D = i) \rightarrow c < \hat{\alpha} \land R \neq \epsilon \land F = i).
\]

If a robot is dispatched and the dispatch signal is changed, it must be reset:

\[
\varphi^*_A,6 = \bigwedge_{i \in [0,m]} \square(\text{clear}(D = i) \rightarrow \bigcirc(D = \epsilon)).
\]

Moreover, dispatching a robot requires the dispatch signal to be clear:

\[
\varphi^*_A,7 = \bigwedge_{i \in [0,m]} \square(\text{raise}(D = i) \rightarrow D = \epsilon).
\]

The counter may only increase by one if a robot has been dispatched:

\[
\varphi^*_A,8 = \square(\bigvee_{\gamma < \hat{\alpha}} (c = \gamma \land \bigcirc(c = \gamma + 1)) \rightarrow \bigvee_{i \in [0,n]} (D = \epsilon \land \bigcirc(D = i))).
\]

Also, the counter may not decrease (unless it is reset), and it may only go up by one at a time. This requirement is captured by \( 0 \leq (\hat{\alpha} + \gamma' - \gamma) \mod \hat{\alpha} \leq 1 \), where \( \gamma \) and \( \gamma' \) are the values of the counter at the current and next step respectively:

\[
\varphi^*_A,9 = \bigwedge_{\gamma \leq \hat{\alpha}, \gamma' \leq \hat{\alpha}} \square(c = \gamma \land \bigcirc(c = \gamma') \rightarrow \text{False}).
\]

This formulation forbids certain changes of the counter \( c \). A positive formulation that ensures that \( c \) only changes in a permissible way can also be used but results in exactly the same synthesized FDS.

If no robot is dispatched, the counter must not change:

\[
\varphi^*_A,10 = \bigwedge_{\gamma \leq \hat{\alpha}} \square(c = \gamma \land \text{stay}(D = \epsilon) \rightarrow \bigcirc(c = \gamma)).
\]

The counter is only reset on a raising edge of the acknowledgement received from FIFO:

\[
\varphi^*_A,11 = \square(c = \hat{\alpha} \land c = 0 \rightarrow \text{raise}(DEQf\_ACK)).
\]

Conversely, if the counter is saturated and a raising edge of the acknowledgement is observed, then the counter is reset:

\[
\varphi^*_A,12 = \square(c = \hat{\alpha} \land \text{raise}(DEQf\_ACK) \rightarrow \bigcirc(c = 0)).
\]

If enough robots have been dispatched to the vertex currently offered by FIFO, then the counter \( c \) is saturated. The allocator must then ask FIFO to offer the location of the next target, if available. For this purpose, the dequeue signal \( DEQf \) is raised if no acknowledgement is currently received:

\[
\varphi^*_A,13 = \square(c = \hat{\alpha} \land \neg DEQf\_ACK \land \neg DEQf \rightarrow \bigcirc DEQf).
\]
5.1. SAR OF STATIONARY TARGET

| Name     | Domain | Description                                               |
|----------|--------|-----------------------------------------------------------|
| $d_j$    | $\{\epsilon, n\}$ | The vertex the robot is dispatched to                  |
| $\text{flag}_j$ | $\mathbb{B}$ | Indicates whether the vertex the robot is on is flagged |

Table 5.3: Environment variables of the robot $R_j$. Note that $d_j$ and $\text{flag}_j$ are both provided by a connector.

The dequeue signal may only go high if the acknowledgement is low:

$$\varphi_{A,14} = \Box(\text{raise}(DEQf) \rightarrow \bigcirc \neg DEQf_\text{ACK}).$$

Also, the dequeue signal must stay high until an acknowledgement is received from FIFOf:

$$\varphi_{A,15} = \Box(\text{clear}(DEQf) \rightarrow DEQf_\text{ACK}).$$

**GR\[1\] Specification and Communication**

The above formulae are composed into the specification of the allocator:

$$\varphi_A = \bigwedge_{\alpha \in [1,4]} \varphi_{A,\alpha} \rightarrow \bigwedge_{\beta \in [1,15]} \varphi_{A,\beta},$$

which can be translated into a GR\[1\] specification.

The communication with FIFOf over the request signal $DEQf$ and the acknowledgement signal $DEQf_\text{ACK}$ follows the four-phase handshake protocol of Section 3.2.3. The assumption $\varphi_{A,4}$ corresponds to III in $M_S$, and the guarantees $\varphi_{A,14}$ and $\varphi_{A,15}$ correspond to VII and VIII respectively. IX is not required for a boolean request signal and XI is indirectly asserted by $\varphi_{A,4}$, $\varphi_{A,3}$ and $\varphi_{A,13}$, given that there are at least $\bar{a}$ robots offered by FIFOr. The property X is not guaranteed from the specification itself, but is required for the communication protocol to be valid. However, model checking the synthesized FDS of the allocator $M_A \models \varphi_A$ reveals that also $M_A \models \Box(\neg DEQf_\text{ACK} \rightarrow \bigcirc DEQf)$. Thus the allocator satisfies the requirements to be a sender in the four-phase handshake protocol.

The allocator communicates with a robot also using the four phase handshake protocol. The request signal that is sent by the allocator is a combination of the index $R$ offered by FIFOr and the index of the target’s vertex in $D$. In the notation of Section 3.2.3, $(r = i) \Leftrightarrow (D = i \land R = j)$ for the request signal to robot $R_j$. The acknowledgement from robot $R_j$ is the signal $r_j$ being cleared. However, $r_j$ is not received as an environment variable by the allocator, so the robot index $R$ offered by FIFOr is used to extract this information: When $R = j$ changes, then robot $j$ has acknowledged the dispatch request. In the notation of Section 3.2.3, $a \Leftrightarrow \text{clear}(R = j)$. Thus $\varphi_{A,2}$ and $\varphi_{A,6}$ correspond to VIII and IX respectively, while X is implied by $\varphi_{A,3}$ and $\varphi_{A,2}$.
5.1. SAR OF STATIONARY TARGET

| Name    | Domain   | Description                                      |
|---------|----------|--------------------------------------------------|
| \( r_j \) | \( \mathbb{B} \) | Indicates whether the robot is ready             |
| \( e_j \) | \( \mathbb{B} \) | Indicates whether the robot is engaged            |
| \( store_j \) | \([\epsilon, n)\) | Stores the current target vertex                  |
| \( cellID_j \) | \([\epsilon, n)\) | Stores the current position of the robot          |
| \( X^j_i, i \in [0, n) \) | \( \mathbb{B} \) | Stores the current position of the robot          |

Table 5.4: System variables of the robot \( R_j \).

5.1.4 Robot Specification

The environment and system variables of the robot \( R_j \) are given in tables Table 5.3 and Table 5.4 respectively. In the following specifications, the sub- and superscripts with the robot’s index \( j \) are omitted.

Assumptions

Initially, it is only known that the robot is not dispatched. This is a consequence of the robot also not being ready when it is started up (see guarantees):

\[ \varphi^e_{R,1} = (d = \epsilon). \]

The allocator must wait for the robot to become ready before it can be dispatched:

\[ \varphi^e_{R,2} = \Box(\text{clear}(d = \epsilon) \rightarrow r). \]

If the allocator sends a dispatch signal, this must remain the same until the robot acknowledges this by clearing its ready signal. Then FIFOr deletes this robot out of its list of robots that are ready, which in turn is (possibly) received by the allocator which only then may clear the dispatch signal:

\[ \varphi^e_{R,3} = \bigwedge_{i \in [0, n)} \Box(\text{clear}(d = i) \rightarrow \neg r). \]

Guarantees

Initially, the robot is not engaged, not ready and has no target stored:

\[ \varphi^s_{R,1} = \neg E \wedge \neg r \wedge store = \epsilon. \]

The robot becomes ready when no target is stored and it is not ready. From this time on, it will turn out that the atomic propositions \( r = \text{True} \) and \( store = \epsilon \) are equivalent, because the robot is ready exactly if no destination is stored:

\[ \varphi^s_{R,2} = \Box(\neg r \wedge store \neq \epsilon \rightarrow \Diamond r). \]

If the robot receives a dispatch signal when it is ready, it fills its target storage with the appropriate destination:

\[ \varphi^s_{R,3} = \bigwedge_{i \in [0, n)} \Box(store = \epsilon \wedge d = i \wedge r \rightarrow \Diamond(store = i)). \]
Moreover, a target may only be stored if the robot is ready. As a result of this, the robot must clear the ready signal \( r \):

\[ \varphi_{\text{R,4}} = \bigwedge_{i \in [0,n)} \square(\text{store} = \epsilon \land \bigcirc(\text{store} = i) \rightarrow r \land d = i \land \neg \bigcirc r). \]

The target store may only be cleared to \( \epsilon \), and only if the robot is at the target location and no flag is detected. Then also the robot must assert that it is ready:

\[ \varphi_{\text{R,5}} = \bigwedge_{i \in [0,n)} \square(\text{clear}(\text{store} = i) \rightarrow \neg \text{flag} \land X_i \land \bigcirc(\text{store} = \epsilon \land r)). \]

When a destination is stored then the robot eventually reaches the required cell:

\[ \varphi_{\text{R,6}} = \square(\text{store} \neq \epsilon \rightarrow \Diamond \bigvee_{i \in [0,n)} (X_i \land \text{store} = i)). \]

If the robot arrives at the correct cell and the flag is still up, then it engages:

\[ \varphi_{\text{R,7}} = \bigwedge_{i \in [0,n)} \square(\text{store} = i \land X_i \land \text{flag} \rightarrow e). \]

If the robot arrives at the correct cell but the flag is already cleared, then the store is cleared, the robot goes back to ready and disengages:

\[ \varphi_{\text{R,8}} = \bigwedge_{i \in [0,n)} \square(\text{store} = i \land X_i \land \neg \text{flag} \rightarrow \bigcirc(r \land \text{store} = \epsilon \land \neg e)). \]

The robot may only engage if it is in the correct cell and the flag is still sensed:

\[ \varphi_{\text{R,9}} = \square(\text{raise}(e) \rightarrow \bigcirc \left( \bigvee_{i \in [0,n)} (X_i \land \text{store} = i) \right) \land \bigcirc \text{flag}). \]

Disengaging requires the target storage to be cleared. Note that this implicitly also includes all that is required to clear the target storage:

\[ \varphi_{\text{R,10}} = \square(\text{clear}(e) \rightarrow \bigvee_{i \in [0,n)} \text{clear}(\text{store} = i)). \]

The robot is free to move as long as it is not engaged.

\[ \varphi_{\text{R,11}} = \bigwedge_{i \in [0,n)} \square(\text{stay}(\)e) \rightarrow (X_i \leftrightarrow \bigcirc X_i)). \]

The allowable moves of the robot depend on the graph \( G \) on which the search is performed. Since \( G \) is given at specification time, the restrictions due to the topology are encoded as presented in Section 4.1.1 and the formula \( \varphi_G \) is included in the robot’s specification.
5.1. SAR OF STATIONARY TARGET

| Name | Domain | Description |
|------|--------|-------------|
| Environment | $E_i$ | [0, $m$] | The number of robots engaged on this cell |
| System | $f_i$ | $\mathbb{B}$ | Indicates whether the cell is flagged |

Table 5.5: Variables of the Cell $C_i$

GR[1] Specification and Communication

The above formulae are composed into the specification of the robot:

$$\varphi_R = \bigwedge_{\alpha \in [1,3]} \varphi_{R,\alpha}^e \implies \varphi_G \land \bigwedge_{\beta \in [1,11]} \varphi_{R,\beta}^s,$$

which can be translated into a GR[1] specification, see Section 3.1.4.

The robot $R_j$ may receive dispatch signals from the allocator via the dispatch signal $d_j$. It acknowledges such a request by clearing its ready signal $r_j$. The allocator does not directly receive $r_j$ but only the signal $R$ from FIFO$_r$, which must be taken into account when reasoning about the correctness of the communication protocols. Thus the acknowledgement signal is equivalent to $r_j = \text{False}$, i.e. when the robot declares that it is no longer ready. Except in the initial state, this is also equivalent to $\text{store}_j \neq \epsilon$. The request signal is equivalent to $d_j \neq \epsilon$. Therefore the guarantee $\varphi_{R,4}^s$ corresponds to XIV, and the combination of $\varphi_{R,3}^s$ and $\varphi_{R,5}^s$ corresponds to XII. However, XV and XIII are not directly guaranteed by the robot but are required in Cases 4 and 5 of the proof of Proposition 3. XIII states that $d_j = \epsilon$ necessarily results in $r_j = \text{True}$. However, since the allocator does not require the liveness assumption I, the proof of Case 5 still works without guaranteeing XIII as follows:

To prove Precondition 1, we deduce II–VI from the conjuncts of the closures of $\varphi_2^R$ and $\varphi_5^R$. II and III follow from $\Box a = \text{True}$ arising from the existential quantification of $a$. IV–VI similarly follow from $\Box r = \epsilon$. To prove Precondition 2, we take $\bar{t} = \text{False}$ in the closure and the result follows trivially.

XV states that $\text{raise}(r_j)$ requires $d_j = \epsilon$. But the proof of Case 4 fails when omitting this guarantee. However, let us consider what would happen if the robot raised $r_j$ when $d_j \neq \epsilon$. The allocator only receives $R$ from FIFO$_r$. If $R = \epsilon$ then $D$ must be $\epsilon$ from $\varphi_{A,3}^s$ and $\varphi_{A,5}^s$, so in this case $d_j \neq \epsilon$ is not possible. Thus at that point $R \neq \epsilon$ and so a robot $R_h$, $h \neq j$ is offered to the allocator in $R$. Thus when $r_j$ is raised, $R$ does not change necessarily, since FIFO$_r$ offers $R = h$ first. However, from the connector proposition for $d_j$, when $R \neq j$ then $d_j = \epsilon$. So also in this case $d_j \neq \epsilon$ is not possible.

5.1.5 Cell and Queue Specifications

Each cell $C_i$ in the architecture in Figure 5.1 has just one environment variable $E_i$, indicating the number of robots engaged on the cell, and one system variable $f_i$, the flag of the cell. These variables are shown in Table 5.5. In the specification, we will again drop the subscripts from the variables and write just $f$ and $E$. 88
5.1. SAR OF STATIONARY TARGET

| Environment | Name | Domain | Description |
|-------------|------|--------|-------------|
| r_j, j ∈ [0, n) | R | ∈ [ε, n) | The index of the robot offered to AMM |

| Table 5.6: Variables of FIFOr. |
|-----------------|-----------------|------------------|
| Environment | Name | Domain | Description |
| f_i, i ∈ [0, m) | DEQf | B | The dequeue request from AMM |
| System | F | ∈ [ε, n) | The index of the vertex offered to AMM |

| Table 5.7: Variables of FIFOf. |
|-----------------|-----------------|------------------|
| Environment | Name | Domain | Description |
| f_i, i ∈ [0, m) | DEQf | B | The dequeue acknowledgement |

A cell has to satisfy only two guarantees. The flag can only be cleared if there are at least \( \bar{\alpha} = 2 \) robots engaged on the cell:

\[
\varphi_{C_1} = \Box(\text{clear}(f) \rightarrow E \geq \bar{\alpha}).
\]

Moreover, we include a liveness property, ensuring that the cell will always be flagged eventually:

\[
\varphi_{C,2} = \Box \Diamond f.
\]

In order for the specification to be realizable, we need to assume that as long as a cell is flagged, the number of robots that are engaged on the vertex associated with that cell will never decrease:

\[
\varphi_{e_1} = \bigwedge_{\gamma \in [0, m)} \Box(f \land E = \gamma \rightarrow \Diamond(E \geq \gamma)).
\]

The FIFO queues FIFOr and FIFOf are not synthesized as FDS’s due to the state space explosion problem, cf. Section 5.1.1. Instead, a thread is implemented that guarantees certain properties to the allocator AMM. In particular, the environment assumptions of AMM on the local system variables controlled by FIFOr and FIFOf must be satisfied. The conceptual environment and system variables of the two queues are shown in Table 5.6 and Table 5.7 respectively.

From \( \varphi_{A,1} \), the allocator requires that initially “DEQf\_ACK = False”, which can easily be implemented in FIFOf. Also, \( \varphi_{A,4} \) requires the dequeue acknowledgement DEQf\_ACK to only be lowered if also the dequeue request DEQf is false, which is similarly straightforward to implement.

There is only one requirement on FIFOr, imposed by \( \varphi_{A,3} \). The robot index R that is offered to AMM may hence only be changed if a robot is dispatched. Since FIFOr keeps track on which robots are ready, and robots may only lower their ready signals r following a dispatch D (cf. the connector (5.3)), this requirement is automatically met.

Lastly, from \( \varphi_{A,3} \), the flag offered to the allocator may only be changed if a dequeue signal DEQf is received by FIFOf or a robot is dispatched. FIFOf
5.1. SAR OF STATIONARY TARGET

Figure 5.2: States of allocator and robot for the stationary target search. The relation for both allocator $AMM$ and robots $R_j$ is quadratic in the number of vertices $n$.

therefore only dequeues a flag either if $DEQf$ is high, or the flag is lowered. By the cell’s guarantee $\psi_C$, and the connector (5.1), a flag can only be lowered if at least $\alpha$ robots are engaged on the corresponding cell. This again is only the case if $D \neq \epsilon$, i.e. a robot is dispatched.

5.1.6 Synthesis

The specifications given above were successfully synthesized using the TuLiP front end. Figure 5.2 shows the number of states of the allocator and the robot when varying the size of the topology and the number of robots available.

The size of the state space of the robot does not depend on the number of robots in the architecture, but the size of the state space of both the allocator and the robots increases quadratically with the number of vertices in the graph representation of the topology. Also, the size of the state space of the allocator increases slightly with the number of robots.

The time for synthesizing the controllers is also polynomial in the size of the topology, as would be expected: The size of the specification increases at most quadratically with $n$, the number of vertices in the graph representing the topology, and the synthesis time is cubic in the size of the specification. By plotting the synthesis time on a logarithmic scale as in Figure 5.3 shows that the complexity must be subexponential. A rough analysis yields polynomial complexity of sixth order in $n$, but this cannot be seen accurately from the available data.
5.1. SAR OF STATIONARY TARGET

Figure 5.3: Time to synthesize allocator and robot for the stationary target search with \( n \) vertices and \( m \) robots. Note that the ordinate is logarithmic but the relation is polynomial.

The timing measurements have been conducted on a 2.4 GHz Intel Core i7 CPU, with 4 GB of working memory. The available memory is likely to also affect the performance because the JTLV implementation uses the JVM Garbage Collector. Note also that the graph presented in this section was obtained for just one random graph for each value of \( n \). Other random graphs give different state space sizes and synthesis times, but the polynomial complexity results still hold.

5.1.7 Discussion

Stationary target search has been successfully simulated for various numbers of robots and vertices. When more than two robots are used, the allocator explicitly requests a new flag to be offered by \( FIFOf \) and dispatches a robot to the corresponding vertex. Thus, with four and more robots several targets can be rescued simultaneously.

However, as is clear from Figure 5.2, the size of the state space grows very fast with the size of the graph. This affects the efficiency of synthesis. The synthesis of the controllers for 20 vertices and 6 robots already took about 15 min, and much longer times are expected for larger graphs. However, space seems to be even a more pressing issue. Larger problems are susceptible to heap overflow due to unsuccessful JVM garbage collection. For example, synthesis of a controller for 16 vertices and 5 robots is not successful when only permitting 1 GB of working memory.

The increase of the state space is mostly due to the added environment variables from the communication protocol. The assumptions on these additional envi-
5.2. MOVING TARGET SEARCH

Environment variables cannot be strengthened sufficiently to reduce the state space further. The communication in this specification already deviates slightly from the four-phase handshake protocol defined in Section 5.2.3 in order to alleviate this problem, as any additional signal like a boolean acknowledgement roughly doubles the state space. Also, beyond a certain graph size, synthesis seems impractical. If this limitation cannot be overcome, then this type of specification seems unsuitable for practical applications.

5.2 Moving Target Search

In contrast to USAR, the targets in WiSAR are considered to be able to move. That means that the stationary target search methods are no longer sufficient, and we implement the distributed multi stage moving target search strategy developed in Section 4.5.3.

There is no central allocator, hence the specification is easier than that of stationary target search. However, as is evident from Section 4.3, justifying that a given number robots with controllers synthesized from the specification reliably find all moving targets requires more effort than with stationary target search.

5.2.1 Communication Structure

As indicated in Section 4.5.1, the robots are organized in a circular structure of authority and communication. A robot \( R_j \) has exactly one slave and exactly one master. Figure 5.4 shows such a cycle. The state variables of the robots are explained in Table 5.8 and Table 5.9.

![Figure 5.4: Four robots communicating in the moving target search architecture. Note that only the local system and environment variables directly relevant for communication are shown.](image)

We use the standard four-phase handshake protocol introduced in Section 3.2.3. Hence no sophisticated arguments as in the previous section on stationary target search are required to justify the correctness of the communication protocols used. Also, the correctness of the composition of all synthesized components is a simple extension of Proposition 2 in Section 3.2.3.
5.2. MOVING TARGET SEARCH

| Name       | Domain              | Description                                      |
|------------|---------------------|--------------------------------------------------|
| $C_{in,j}$ | $(\epsilon, n)$    | Vertex the robot is sent to by its master         |
| $ackin_j$  | $\mathbb{B}$        | Acknowledgement received from its slave           |
| $cst_j$    | $\{cl, pc, cr, sv\}$| State of the vertex on which the robot is         |

Table 5.8: Environment Variables of Robot $R_j$

| Name       | Domain              | Description                                      |
|------------|---------------------|--------------------------------------------------|
| $M_j$      | $\{gu, cs, sp, sl\}$| Mode of the robot                                |
| $store_j$  | $[\epsilon, n)$    | Vertex stored for the clearing mode               |
| $cellID_j$ | $[0, n)$            | Current position of the robot                     |
| $X^i_j$, $i \in [0, n)$ | $\mathbb{B}$ | True if robot $R_j$ is in position $i$ |
| $c_j$      | $[\epsilon, \gamma_{\text{max}}]$ | Counter in clearing mode |
| $cnt_j$    | $[0, |p|)$          | Counter in search modes                           |
| $mcnt_j$   | $[0, |p|)$          | Maximum counter value in search modes             |
| $ackout_j$ | $\mathbb{B}$        | Acknowledgement sent to master                    |
| $C_{out,j}$| $(\epsilon, n)$    | Vertex to send the robot’s slave to               |

Table 5.9: System Variables of Robot $R_j$

5.2.2 Robot Specifications

Similar to the stationary target search, the global specification must state that any target is eventually found on the given topology. Since the search strategies developed in Section 4.5 ensure that the graph is cleared, this global requirement is necessarily satisfied, given that the architecture contains sufficiently many robots. We therefore only state the specifications that are synthesized into robots implementing a winning strategy for the cops. It is then easy to extend this to include sensors to detect targets and react appropriately (e.g. by engaging).

In the following specifications, the subscripts and superscripts indicating the index of the robot are omitted for notational convenience.

Guarantees: Topology

Similar to the stationary target search, the allowable moves of the robot have to be defined in the guarantee part of the specification. The restrictions due to the topology are encoded as presented in Section 4.1.1, and the formula $\varphi_G$ encoding the topology is included in the robot’s specification.

Note that the targets are restricted to move on the same topology. However, since they are assumed to be able to move with infinite speed, and the graph is strongly connected, the robot cannot assume any restriction on the movement of the targets.
5.2. MOVING TARGET SEARCH

Guarantees: Guarding Mode

In the guarding mode $M = \text{gu}$, the robot has to stay at the current vertex unless a command from its master is received via $C_{\text{in}}$:

$$\varphi_{M,1}^g = \bigwedge_{i \in [0,n)} \Box(M = \text{gu} \land C_{\text{in}} = c \land X_i \rightarrow \bigcirc(X_i \land M = \text{gu})).$$

However, if a command is received, then the clearing mode $M = \text{sl}$ has to be entered, setting up the store and counter appropriately:

$$\varphi_{M,2}^g = \bigwedge_{i \in [0,n)} \Box(M = \text{gu} \land C_{\text{in}} = c \rightarrow \bigcirc(M = \text{sl} \land \text{store} = i \land c = \epsilon)).$$

If the guarding mode is entered, then the robot’s slave is instructed to clear all out-edges of the current vertex $v_i$ by asserting the index $i$ on $C_{\text{out}} = i$. However, if the guarding mode is entered because the counter $c$ is saturated, this means that no vertex was found in $M = \text{cs}$ or $M = \text{sp}$. In this case we know that the graph is cleared and so $C_{\text{out}}$ stays at $\epsilon$.

$$\varphi_{M,3}^g = \bigwedge_{i \in [0,n)} \bigwedge_{\gamma \in [0,|p|)} \Box(M = \text{cs} \land \bigcirc(M = \text{gu}) \land \bigcirc(X_i) \land \text{cnt} = \gamma \land \text{mcnt} \neq \gamma \rightarrow \bigcirc(C_{\text{out}} = i)).$$

Guarantees: Searching Modes

The searching modes $M = \text{cs}$ and $M = \text{sp}$ can be specified together, as they differ only slightly by the state of the vertex the robot is searching for and the mode that is entered if such a vertex is encountered.

First, when entering a searching mode, the counters must be initialized correctly:

$$\varphi_{M,4}^g = \bigwedge_{i \in [0,n)} \Box(X_i \land (\text{raise}(M = \text{cs}) \lor \text{raise}(M = \text{sp})))$$

$$\rightarrow \bigcirc(\text{cnt} = \gamma_i \land \text{mcnt} = \text{nextval}(\gamma_i) \land X_i),$$

where $\gamma_i$ is the index of the first occurrence of the vertex $v_i$ in the path $p$. Note that $\text{nextval}(\gamma) = (\gamma + 1 + |p|) \mod |p|$ as defined in Section 4.2.1 before (4.2).

The robot moves along the path $p$ one vertex at a time, making sure that the counter $\text{cnt}$ increases at every transition:

$$\varphi_{M,5}^g = \bigwedge_{\gamma \in [0,|p|)} \Box(X_{p(\gamma)} \land \text{cnt} = \gamma \land \text{mcnt} \neq \gamma \land \text{cnt} = \gamma \land c = \epsilon \land X_i)$$

$$\rightarrow \bigcirc(\text{cnt} = \text{nextval}(\gamma) \land X_{p(\text{nextval}(\gamma))}).$$

If a start vertex has been found, then enter the guarding mode $M = \text{gu}$:

$$\varphi_{M,6}^g = \bigwedge_{i \in [0,n)} \Box(M = \text{cs} \land \bigcirc(\text{cnt} = \text{sv}) \land X_i \rightarrow \bigcirc(M = \text{gu} \land c = \epsilon \land X_i)).$$
Similarly, if a partially cleared vertex $v$ has been found, then enter the clearing mode $M = \text{sl}$, initialized to clear the out-edges of $v$:

$$\varphi^s_{M,7} = \bigwedge_{i \in [0,n]} (M = \text{sp} \land \Box(cst = \text{pc}) \land X_i \rightarrow \Box(M = \text{sl} \land c = \epsilon \land \text{store} = i \land X_i)).$$

Also, if the maximum counter value is reached in $M = \text{sp}$, then there is no partially cleared vertex and the search for a start vertex $M = \text{cs}$ is entered:

$$\varphi^s_{M,8} = \bigwedge_{\gamma \in [0,|p|]} (M = \text{sp} \land \text{cnt} = \gamma \land \text{mcnt} = \gamma \land \Box(cst \neq \text{pc}) \rightarrow \Box(M = \text{cs})).$$

Note that the values of the counters are reset to the correct values necessarily by the guarantee $\varphi^s_{M,4}$. Similarly, if the maximum counter value is reached in $M = \text{cs}$, then there is no contaminated (start) vertex and the graph must necessarily be cleared and the robot becomes dormant in guarding mode $M = \text{gu}$. Note that an additional mode can be introduced in this case that allows the robots to move arbitrarily on the graph, but here only the concept of moving target search should be introduced. The corresponding guarantee is:

$$\varphi^s_{M,9} = \bigwedge_{\gamma \in [0,|p|]} (M = \text{cs} \land \text{cnt} = \gamma \land \text{mcnt} = \gamma \land \Box(cst \neq sv) \rightarrow \Box(M = \text{gu})).$$

Now a few guarantees are required that prevent the counters from changing when they should not (so called frame axioms). The counter of the position in the path $p$, $\text{cnt}$ may only increase until it overflows, corresponding to the path closing in a cycle.

$$\varphi^s_{M,10} = \Box(stay(M = \text{cs}) \lor stay(M = \text{sp}) \rightarrow \bigvee_{\gamma \in [0,|p|]} (\text{cnt} = \gamma \land \Box(\text{cnt} = \text{nextval}(\gamma)))).$$

Similarly, the value stored as maximum for the counter may not change:

$$\varphi^s_{M,11} = \Box(stay(M = \text{cs}) \lor stay(M = \text{sp}) \rightarrow \bigvee_{\gamma \in [0,|p|]} (\text{mcnt} = \gamma \land \Box(\text{mcnt} = \gamma))).$$

Since $\text{cnt}$ and $\text{mcnt}$ are only used in $M = \text{cs}$ or $M = \text{sp}$, they are set to zero otherwise:

$$\varphi^s_{M,12} = \Box(\neg(M = \text{cs} \lor M = \text{sp}) \rightarrow \text{cnt} = 0 \land \text{mcnt} = 0).$$

Two more guarantees are required to prevent the robot from leaving the searching mode if no start vertex or partially cleared vertex was encountered or the counter has not reached its maximum value:

$$\varphi^s_{M,13} = \Box(\neg(\text{clear}(M = \text{cs}) \rightarrow \bigvee_{\gamma \in [0,|p|]} (\text{cnt} = \gamma \land \text{mcnt} = \gamma) \land \Box(cst = sv)),
\varphi^s_{M,14} = \Box(\neg(\text{clear}(M = \text{sp}) \rightarrow \bigvee_{\gamma \in [0,|p|]} (\text{cnt} = \gamma \land \text{mcnt} = \gamma) \land \Box(cst = pc)).$$
Guarantees: Clearing Mode

The clearing mode is the most involved mode, but also the mode that is responsible for actual progress in clearing edges: no edges can be cleared by a robot that is not in mode \( M = \text{sl} \).

First, on entering clearing mode, the index of a vertex is written to \( \text{store} \) so that the robot knows for which vertex the out-edges must be cleared. This can only happen when the robot is guarding and called by its master to move, or when the robot has found a partially cleared cell that it is now about to clear:

\[
\varphi_{M,15}^s = \bigwedge_{i \in [0,n)} \Box \left( \begin{array}{l}
\text{store} = \epsilon \land \bigcirc \left( \text{store} = i \right) \rightarrow \\
(C_{\text{in}} = i \land M = \text{gu}) \lor (M = \text{sp} \land \bigcirc (\text{cst} = \text{pc} \land X_i))
\end{array} \right).
\]

Also, \( \text{store} \) may only change from \( \epsilon \) if also the counter \( c \) is reset to \( \epsilon \) or alternatively if the robot enters the clearing mode from a search for a partially cleared vertex:

\[
\varphi_{M,16}^s = \Box \left( \begin{array}{l}
\text{clear} \left( \text{store} = \epsilon \right) \rightarrow \bigcirc (\epsilon = c) \land M = \text{gu} \land \bigcirc (M = \text{sl}) \lor \\
(M = \text{sp} \land \bigcirc (\text{cst} = \text{pc}) \land \bigcirc (M = \text{sl}))
\end{array} \right).
\]

Moreover, if \( \text{store} \neq \epsilon \), then it may only change to \( \epsilon \), i.e. it is not permitted to immediately change from sliding down the out-edges of one vertex to sliding down the out-edges of another. Such a change also requires the counter \( c \) to be saturated, indicating that all out-edges of the vertex held in \( \text{store} \) are successfully cleared:

\[
\varphi_{M,17}^s = \bigwedge_{i \in [0,n)} \Box \left( \begin{array}{l}
\text{clear} \left( \text{store} = i \right) \rightarrow \bigcirc (\epsilon = c) \land c = \bar{\gamma}_i \land \bigcirc (c = \epsilon)
\end{array} \right).
\]

Conversely, if the counter \( c \) is saturated, then the robot has successfully cleared all out-edges of the vertex held in \( \text{store} \) and thus enters the search for a partially cleared vertex:

\[
\varphi_{M,18}^s = \bigwedge_{i \in [0,n)} \Box \left( \begin{array}{l}
\text{store} = \epsilon \land c = \bar{\gamma}_i \rightarrow \bigcirc \left( \text{store} = \epsilon \land c = \epsilon \land M = \text{sp} \right)
\end{array} \right).
\]

The counter \( c \) is always diligently set to \( \epsilon \) when the clearing mode is entered (see \( \varphi_{M,26}^s \)), representing that the robot first has to move to the vertex held in \( \text{store} \). This latter requirement is expressed using the following property:

\[
\Box \left( \text{store} \neq \epsilon \land c = \epsilon \rightarrow \bigcirc \bigvee_{i \in [0,n)} \left( \text{store} = i \land X_i \right) \right). \tag{5.4}
\]

Note that we write \( \Box (\pi \rightarrow \bigcirc \bigvee_{i \in [0,n)} (\text{store} = i \land X_i)) \) rather than the more intuitively obvious \( \bigwedge_{i \in [0,n)} \Box (\pi \land \text{store} = i \rightarrow \bigcirc X_i) \). This is because the latter introduces \( n \) response formulae, each of which requires a trigger variable to be added to the system variables, which would multiply the potential number of states by \( 2^n \). Using the former contracted response property only requires one trigger and thus the synthesized FDS has fewer states.
While the robot moves to the vertex held in \( \text{store} \), the counter \( c \) must remain at \( \varepsilon \):

\[
\varphi_{M,19}^s = \bigwedge_{i \in [0,n]} \Box (\text{store} = i \land c = \varepsilon \land \neg X_i \rightarrow \bigcirc (c = \varepsilon)).
\]

On reaching the vertex, the robot sets the counter to zero. It also stays in the vertex so that the state of the robot is consistent:

\[
\varphi_{M,20}^s = \bigwedge_{i \in [0,n]} \Box (\text{store} = i \land c = \varepsilon \land X_i \rightarrow \bigcirc (c = 0 \land X_i)).
\]

Given that the robot is in the vertex held in \( \text{store} \) and the counter \( c \) has a value not equal to \( \varepsilon \), an out-edge can be cleared. The corresponding property has already been introduced in Section 4.5.3:

\[
\varphi_{M,21}^s = \bigwedge_{i \in [0,n]} \bigwedge_{\gamma \in [0,\bar{\gamma}_i]} \Box (\text{store} = i \land c = \gamma \land X_i \rightarrow \bigcirc (X_{w_i} \land c = \gamma + 1)).
\]

Recall that \( \bar{\gamma}_i \) is the number of out-edges of the vertex \( i \) and \( w_{\gamma} \) is the (unique) vertex such that there is an edge from \( i \) to \( w_{\gamma} \) and \( e \) is the \( \gamma \)th edge to be cleared by the robot. After clearing the edge, the robot must move back to the vertex held in \( \text{store} \), which is expressed by the guarantee

\[
\Box \left( \bigvee_{i \in [0,n]} \text{store} = i \land \bigvee_{\gamma \in [0,\bar{\gamma}_i]} (c = \gamma + 1 \land X_{w_i}) \rightarrow \bigcirc \bigvee_{i \in [0,n]} (\text{store} = i \land X_i) \right).
\]

This again is a response property, which would require a trigger to be introduced. Instead of having both response properties (5.4) and (5.5), we combine them into one guarantee:

\[
\varphi_{M,22}^s = \Box (\text{store} \neq \varepsilon \land c = \varepsilon) \lor \bigvee_{i \in [0,n]} (\text{store} = i \land \bigvee_{\gamma \in [0,\bar{\gamma}_i]} (c = \gamma + 1 \land X_{w_i})) \rightarrow
\]

\[
\bigcirc \bigvee_{i \in [0,n]} (\text{store} = i \land X_i).
\]

This is of course only possible since the right hand side of the implication \( \rightarrow \) is the same in both (5.4) and (5.5).

Again, the counter \( c \) must be restricted. It may only increase by one if an edge is cleared:

\[
\varphi_{M,23}^s = \bigwedge_{\gamma \in [0,\alpha]} \Box (c = \gamma \land \Box (c = \gamma + 1) \rightarrow \bigvee_{i \in [0,\bar{\gamma}_i]} (\text{store} = i \land X_i \land \bigcirc X_{w_i})).
\]

Also, the counter may not decrease unless it is reset. Moreover, it may only increase by one:

\[
\varphi_{M,24}^s = \bigwedge_{i \in [0,n]} \Box (\text{store} = i \rightarrow \bigvee_{\gamma_1 \in [\varepsilon,\bar{\gamma}_i]} \bigvee_{\gamma_2 \in [\varepsilon,\bar{\gamma}_i]} \text{increase}(c, \gamma_1, \gamma_2,i)).
\]
5.2. MOVING TARGET SEARCH

where \( \text{increase}(c, \gamma_1, \gamma_2, i) \) is a formula that only holds if the counter changes by an admissible amount, which is defined as follows:

\[
\text{increase}(c, \gamma_1, \gamma_2, i) = \begin{cases} 
  c = \gamma_1 \land (c = \gamma_2) & \text{if } \gamma_2 = \gamma_1 + 1 \\
  \text{or } \gamma_1 = \bar{\gamma}_i \land \gamma_2 = \epsilon \\
  \text{or } \gamma_1 = \gamma_2 & \text{otherwise}
\end{cases}
\]

The counter may only be reset to \( \epsilon \) if all out-edges have been cleared:

\[
\varphi_{M,25}^s = □\left(\text{raise}(c = \epsilon) \rightarrow \bigvee_{i \in [0,n]} (\text{store} = i \land c = \bar{\gamma}_i)\right).
\]

If the robot is not in the clearing mode, then the counter \( c \) and \( \text{store} \) are not used, and they are therefore set to \( \epsilon \):

\[
\varphi_{M,26}^s = □(M \neq \text{sl} \rightarrow c = \epsilon \land \text{store} = \epsilon).
\]

Finally, the robot has to remain in the clearing mode as long as there are still out-edges of the vertex held in \( \text{store} \) to be cleared.

\[
\varphi_{M,27}^s = □(M = \text{sl} \land \neg \bigwedge_{i \in [0,n]} (\text{store} \neq i \lor c = \bar{\gamma}_i) \rightarrow □(M = \text{sl})).
\]

**Communication**

The robots communicate with each other using the four-phase handshake protocol introduced in Section 3.2.3. Each robot communicates with its slave and its master as receiver and sender respectively, see Section 4.5.1. The sender specifications, i.e. for the communication of the robot as master, consist of the following guarantees:

\[
\begin{align*}
\text{VII} & \quad \varphi_{M,28}^s = □(M = \text{gu} \land \text{clear}(C_{\text{out}} = \epsilon) \rightarrow \neg \text{ackin}), \\
\text{VIII} & \quad \varphi_{M,29}^s = □(M = \text{gu} \land \text{raise}(C_{\text{out}} = \epsilon) \rightarrow \text{ackin}), \\
\text{X} & \quad \varphi_{M,30}^s = □(M = \text{gu} \land \text{ackin} \rightarrow □(C_{\text{out}} = \epsilon)),
\end{align*}
\]

and of the following assumptions:

\[
\begin{align*}
\text{II} & \quad \varphi_{M,1}^e = □(\text{raise}(\text{ackin}) \rightarrow C_{\text{out}} \neq \epsilon), \\
\text{III} & \quad \varphi_{M,2}^e = □(\text{clear}(\text{ackin}) \rightarrow C_{\text{out}} = \epsilon).
\end{align*}
\]

Note however, that the assumption \( □(C_{\text{out}} = \epsilon \rightarrow \Diamond \neg \text{ackin}) \) corresponding to I is not included, since then the specification would not be synthesizable. This is because the rest of the specification implicitly includes \( □(\pi(\text{ackin}) \rightarrow \Diamond \pi'(C_{\text{out}})) \) for some propositional formulae \( \pi \) and \( \pi' \). However, leaving out an assumption does not render the protocol incorrect (while leaving out a guarantee would). The guarantee IX is not necessary, since the request signal \( C_{\text{out}} \) is boolean and a specification \( □(C_{\text{out}} \land \Diamond \neg C_{\text{out}} \rightarrow \Diamond \neg C_{\text{out}}) \) is redundant. The conjunct \( M = \text{gu} \) in the antecedents of the guarantees is necessary to ensure realizability since \( C_{\text{out}} \) must be permitted to be asserted when guarding mode is entered (cf. guarantee \( \varphi_{M,3}^e \)).
The receiver specifications, i.e. for the communication of the robot as a slave consist of the following guarantees:

\begin{align*}
\text{XII} & \quad \phi_{s,M,31} = \Box(C_{in} \neq \epsilon \rightarrow 
\text{ackout}), \\
\text{XIII} & \quad \phi_{s,M,32} = \Box(C_{in} = \epsilon \rightarrow \neg \text{ackout}), \\
\text{XIV} & \quad \phi_{s,M,33} = \Box(\text{raise}(\text{ackout}) \rightarrow C_{in} \neq \epsilon), \\
\text{XV} & \quad \phi_{s,M,34} = \Box(\text{clear}(\text{ackout}) \rightarrow C_{in} = \epsilon),
\end{align*}

and of the following assumptions:

\begin{align*}
\text{VI} & \quad \phi_{e,M,5} = \Box(\text{raise}(C_{in} = \epsilon) \rightarrow \text{ackout}), \\
\text{VII} & \quad \phi_{e,M,6} = \Box(\text{clear}(C_{in} = \epsilon) \rightarrow \neg \text{ackout}).
\end{align*}

Note again that the assumption \(\Box(\text{ackout} \rightarrow \Diamond(C_{in} = \epsilon))\) corresponding to V is not included due to problems with synthesizability. But since all guarantees are included, the communication protocol works as proved above in Proposition 2 in Section 3.2.3.

Assumptions

While the specification above is perfectly realizable, due to the large number of possible valuations of the environment variables \(|C_{in}| \times |\text{ackin}| \times |\text{cst}| = 8n\) it is beneficial to make assumptions on the environment to reduce the number of transitions that have to be included in the synthesized FDS (see Section 2.2.2).

Since the state of a vertex is only required in the searching modes \(M = \text{cs}\) and \(M = \text{sp}\), we can artificially set \(\text{cst}\) to some arbitrary value in these modes. Then the robot can make the following two assumptions on the environment:

\begin{align*}
\phi_{M,5} &= \Box(M \neq \text{cs} \land M \neq \text{sp} \rightarrow \Diamond(\text{cst} = \text{cr})), \\
\phi_{M,6} &= \Box(\text{stay}(M \neq \text{cs} \land M \neq \text{sp}) \rightarrow \Diamond(\text{cst} = \text{cr})).
\end{align*}

Here we chose to set \(\text{cst} = \text{cr}\). Note that this has to be guaranteed by the global storage of the graph state, cf. Section 4.5.2.

5.2.3 Synthesis

The full specification of the robots results is the following

\[ \varphi_M = \bigwedge_{1 \leq \alpha \leq 6} \varphi_{M,\alpha} \rightarrow v_G \land \bigwedge_{1 \leq \beta \leq 346} \varphi_{M,\beta}, \]

which can be translated into a GR[1] specification and hence synthesized with TLV.

The robot specifications given above were successfully synthesized using the TuLiP front end. For varying numbers of vertices in the topology, nine \(^1\) random graphs were generated each in order to investigate the state space size and

\(^1\)For \(n = 11\) only four samples were calculated.
5.2 MOVING TARGET SEARCH

Figure 5.5: States of the robots for the moving target search. The mean and the standard deviation of nine samples are shown in bold and dashed lines respectively.

synthesis time, plotted in Figure 5.5 and Figure 5.6. The resulting mean and standard deviation are presented by bold and dashed lines respectively.

As with the controllers for stationary target search, the size of the state space depends quadratically on the size of the graph $n$. The synthesis time for the moving target search controllers is polynomial as well, since the curve on the logarithmic plot in Figure 5.6 is sublinear. However, for a given $n$ the state space size of the controllers exceeds those in stationary target search. The same is true for the time they take to synthesize (cf. Figure 5.2 and Figure 5.3).

5.2.4 Simulation

We show a simulation of the moving target search on a small graph $G$ with five vertices. The cop number is $c(G) = 3$, and so we use three robots. The specification is synthesized first and then the resulting FDS is executed in three different threads, corresponding to three robots on the graph with controllers $M_{c_0}, M_{c_1}$ and $M_{c_2}$. The communication between the robots is realized using a shared data structure holding the valuations of the global state variables $V$ of the composite FDS $M_\phi = M_{c_0} \mid M_{c_1} \mid M_{c_2}$.

As explained in Section 5.3, all robots are initialized to $M = gu$, except one, which is initialized to $M = cs$. Figure 5.7(a) shows the initial configuration on the graph. Solid edges are contaminated while dashed edges are cleared. A vertex is black, red, green and dashed if it is contaminated, critical, partially cleared or cleared, respectively.
5.2. MOVING TARGET SEARCH

Figure 5.6: Time to synthesize allocator and robot for the moving target search with $n$ vertices. The mean and the standard deviation of nine samples are shown in bold and dashed lines respectively. Note that the ordinate is logarithmic.

The execution of the asynchronous composition of $\mathcal{M}_{\emptyset_0}$, $\mathcal{M}_{\emptyset_1}$, and $\mathcal{M}_{\emptyset_2}$ results in a sequence of states $s_0 \leq s_1 \leq \ldots \models \mathcal{M}_G$. Significant points in this sequence are shown in Figure 5.7. It can be seen that the graph is successfully cleared with the given number of robots. The specifications do not state how the target is detected, in which case the moving target search would have ended with the target being found (if one exists).

5.2.5 Discussion

Again, as in the case of the stationary target search, the size of the state space increases quadratically with the number of vertices in the graph. However, the number of robots does not influence the size of the state space, since each robot has exactly one master and one slave and no further negotiation is required. Still, due to the required additional variables for communication, the synthesized FDS's are even larger than those in stationary target search.

In the searching modes, the circular search introduced in Section 4.2.2 is used. This results in a rather inefficient moving target search implementation. However, the robots only have local information, so when global properties such as “for all partially cleared vertices” need to be implemented in such a distributed way, a tradeoff between information and efficiency is introduced. Note also that by only using $V_G^L$, the heuristics $H$ uses local information to decide on the start vertex.

When a robot is guarding a vertex, it cannot move. In this specification of moving target search, there is always only one robot not in guarding mode un-
5.2. MOVING TARGET SEARCH

Figure 5.7: Simulation result of moving target search on a graph with five vertices and three robots.

(a) $s_0$: Initial configuration.

(b) $s_1$: Robot $R_0$ starts moving.

(c) $s_6$: Robot $R_0$ moves onto the starting vertex $v_4$.

(d) $s_9$: Robot $R_0$ realizes that $v_4$ is a starting vertex and goes to $M = gu$, calling its slave $R_1$.

(e) $s_{24}$: Robot $R_1$ goes to $M = sl$ and arrives at $v_4$.

(f) $s_{26}$: Robot $R_1$ clears the edge from $v_4$ to $v_0$. $v_4$ becomes critical.
5.2. MOVING TARGET SEARCH

Figure 5.7: Simulation result of moving target search on a graph with five vertices and three robots.
5.2. MOVING TARGET SEARCH

(m) \( s_{136} \): \( R_1 \) discovered no partially cleared vertex and thus goes to \( M = cs \) to search for the next starting vertex.

(n) \( s_{139} \): Coincidentally, the vertex \( R_1 \) is already the starting vertex, so \( R_1 \) enters guarding mode \( M = gu \) without moving.

(o) \( s_{143} \): \( R_2 \) is the slave of \( R_1 \) and enters \( M = sl \) to clear the out-edges of \( v_1 \).

(p) \( s_{159} \): All out-edges of \( v_3 \) are cleared, and the graph is already cleared, but the robots continue moving until all edges are cleared.

(q) \( s_{183} \): All edges are cleared after \( R_2 \) has cleared \( v_0 \) which was partially cleared in \( s_{159} \). \( R_2 \) is still in \( M = sp \) after clearing \( v_0 \).

(r) \( s_{213} \): After \( R_2 \) has unsuccessfully searched for partially cleared vertices and another starting vertex, it also enters \( M = gu \).

Figure 5.7: Simulation result of moving target search on a graph with five vertices and three robots.
5.2. MOVING TARGET SEARCH

less the graph is already cleared. This might seem inefficient, as only one robot can move. However, the guarding robots are playing an important part in the search; if they move, recontamination could revert the progress of previous sliding moves.

One reason why the moving target search strategy can be implemented in such a way is because the global storage (cf. Section 4.5.2) allows the robots to not retain any historical knowledge of the graph state. Also, the heuristics \( \mathcal{H} \) is implemented in the global storage, so that the start vertex is offered to a robot \( R_j \) simply by letting \( \text{cst}_j = \text{sv} \). In a real-world implementation this has to be taken into account.
5.2. MOVING TARGET SEARCH
Chapter 6

Conclusion

In this thesis, a framework for specifying and reasoning about communicating and cooperating SAR robots is introduced, and the synthesis of controllers from their specifications is demonstrated. We further define and verify a communication protocol, allowing reliable asynchronous transmission of data between the robots.

The methods are demonstrated by successfully synthesizing and simulating controllers for robots performing centrally coordinated SAR of stationary targets, and for robots performing distributed search of moving targets. This demonstrates that it is possible to automatically generate discrete controllers for coordinated SAR operations that are mathematically proven to achieve their high-level goals.

6.1 Observations

The controllers cannot ensure bounds on the time it takes to eventually find a target. This is limitation inherent in the fragment of the specification language that we use and the synthesis method. However, fast searching is paramount, as with increasing time the chances of successful rescue dwindle, which is exacerbated in WiSAR, where it is usually necessary to increase the search radius with time.

With respect to the expected time to find a target, there is a tradeoff between the accuracy and speed of effective search. These two factors are also called detection and coverage [31]. With greater accuracy the search takes longer and the probability of a SAR operation being successful decreases. In typical SAR operations, exhaustive search is only resorted to if other more effective search strategies have failed to yield useful information.

The discrete controllers are synthesized to satisfy their specifications with mathematically proven certainty. However, this requirement might be too rigorous for uncertain environments, compromising robustness in the case of unexpected scenarios or failures. For example, a change in the topology would render a controller unusable and cause the robot to stop operating. Also, it might be
sufficient to guarantee that targets are found only with very high probability, or almost surely, while ensuring a faster overall operation.

With this type of controllers, the tradeoff between detection and coverage is completely ignored, and the search is always exhaustive. The accuracy of detection depends on the sensors being able to reliably detect targets. No such sensor exists, and sensors with high false-alarm rates lead to misallocating rescue robots that could be used otherwise to improve coverage.

6.2 Viability of Approach

Two key themes limiting the versatility of the proposed methods throughout this thesis are the computational complexity of the synthesis methods and the size of the state space. It may be objected that all information required for the controller is already encoded in the specification, and that the synthesis is wasteful both in terms of time and space. A program in a procedural language rather than an FDS could be used instead with the same effect but with lower space requirements. However, for complex specifications, program validation can likewise be a lengthy task.

In particular for the search strategies, algorithms have been developed that have proven to yield the same results as the tediously specified FDS’s. It might not be viable to translate an existing algorithm into LTL and then synthesizing it back into an executable FDS. However, development of rule-based controllers or incrementally subjecting algorithms to more requirements can be handled straightforwardly with LTL specifications (as long as the specification remains realizable), e.g. introducing a new traffic rule when controlling an autonomous vehicle for urban traffic.

6.3 Future Work

Several improvements and extensions that are relevant for the specification and synthesis of controllers from LTL can be made. In SAR operations, it is often required that there are agents designated to particular tasks, e.g. robots specialized for searching large areas, searching on a small scale and extracting victims, or providing medical relief. This heterogeneity can be exploited for more efficient SAR strategies\[31\]. Moreover, a distribution of the resources can be included, such as partitioning the topology among several teams of robots.

The currently implemented search strategies are effective, but inefficient. This can be partially alleviated by the above improvements. Another approach would be to allow more robots than the cop number and implement fast search strategies that require each edge to only be traversed once by a robot\[55\]. Also, efficient search methods that rely on probabilistic information on the environment can be synthesized from probabilistic temporal logic specifications\[7, 28\].
The controller specifications are small compared to the FDS’s resulting from synthesis with TLV. It would be beneficial to exploit the compactness of a specification and perform synthesis on-the-fly instead of generating a static FDS. Such dynamic approaches are already used for model checking, e.g. by synthesizing deterministic monitors [6]. Dynamic synthesis is still in its infancy, but receding horizon, on-the-fly and compositional methods are already being developed [26, 90].

Finally, Chung et al. [17] state that

“due to poor reliability and the number of possible failures, few multi-robot systems have been tested on the scale necessary to demonstrate pursuit-evasion in complex environments.”

This thesis shows that theoretically the synthesis of controllers for SAR robots is viable, but neither the technology of robots is yet advanced enough, nor is controller synthesis explored to the extent that large-scale complex SAR scenarios can be implemented based on the methods that we present. The small existing sample implementations do not yet allow reliable communication between the robots [46]. Given the promises in robot-assisted SAR and the complexity reduction through synthesis, it would be a rewarding endeavor to test the presented methods on hardware.
Bibliography

[1] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.

[2] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–535, May 1995.

[3] L. Alfaro. Formal verification of probabilistic systems. Technical report, Stanford, CA, USA, 1998.

[4] M. Antoniotti and B. Mishra. Discrete events models + temporal logic = supervisory controller: Automatic synthesis of locomotion controllers. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 1, pages 1441–1446, 1995.

[5] R.C. Arkin, T. Balch, and E. Nitz. Communication of behavioral state in multi-agent retrieval tasks. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 3, pages 588–594, 1993.

[6] R. Armoni, D. Korchemny, A. Tiemeyer, M. Y. Vardi, and Y. Zbar. Deterministic dynamic monitors for linear-time assertions. In *Proceedings of the International ICSC Symposium on Multi-Agents and Mobile Agents in Virtual Organizations and E-Commerce*, pages 1–20. Springer-Verlag, 2006.

[7] C. Baier, M. Größer, M. Leucker, B. Bollig, and G. Ciesinski. Controller synthesis for probabilistic systems. In J.J. Lévy, E.W. Mayr, and J.C. Mitchell, editors, *Proceedings of IFIP International Conference on Theoretical Computer Science*, pages 493–506. Kluwer, 2004.

[8] S. Belardo, K.R. Karwan, and W.A. Wallace. Managing the response to disasters using microcomputers. *Interfaces*, 14(2):29–39, 1984.

[9] C. Belta, A. Bicchi, M. Egerstedt, E. Frazzoli, E. Klavins, and G. J. Pappas. Symbolic planning and control of robot motion: State of the art and grand challenges. *IEEE Robotics and Automation Magazine*, 14(1):61–70, 2007.

[10] J.G. Blitch. Artificial intelligence technologies for robot assisted urban search and rescue. *Expert Systems with Applications*, 11(2):109–124, 1996.

[11] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weighhofer. Specify, compile, run: Hardware from PSL. *Electronic Notes in Theoretical Computer Science*, 190(4):3–16, 2007.

[12] A. Bolotov, A. Basukoski, O. Grigoriev, and V. Shangin. Natural deduction calculus for linear-time temporal logic. In *JELIA*, pages 56–68, 2006.

[13] Richard B. Borie, Craig A. Tovey, and Sven Koenig. Algorithms and complexity results for graph-based pursuit evasion. *Auton. Robots*, 31(4):317–332, 2011.

[14] M. Campbell, M. Egerstedt, J.P. How, and R.M. Murray. Autonomous driving in urban environments: approaches, lessons and challenges. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 368:4649–4672, 2010.
[15] J. Casper and R.R. Murphy. Human-robot interactions during the robot-assisted urban search and rescue response at the World Trade Center. IEEE Transactions on Systems, Man and Cybernetics, 33(3):367–385, 2003.

[16] K. Chatterjee, M. Jurdziński, and T.A. Henzinger. Quantitative stochastic parity games. In Proceedings of the 15th annual ACM-SIAM symposium on Discrete algorithms, pages 121–130, 2004.

[17] T. Chung, G. Hollinger, and V. Isler. Search and pursuit-evasion in mobile robotics. Autonomous Robots, 31:299–316, 2011.

[18] E.M. Clarke, O. Grumberg, and D. Peled. Model checking. MIT Press, 1999.

[19] H.G. Daepp and W.J. Book. Modeling and simulation of a pneumatically-actuated rescue robot. In Proceedings of the 52nd National Conference on Fluid Power, 2011.

[20] L. de Alfaro, T. A. Henzinger, and F. Y. C. Mang. The control of synchronous systems. In CONCUR, pages 458–473, 2000.

[21] E.A. Emerson and J.Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In Proceedings of the fourteenth annual ACM symposium on Theory of Computing, pages 169–180. ACM, 1982.

[22] E.A. Emerson and J.Y. Halpern. “sometimes” and “not never” revisited: on branching versus linear time temporal logic. Journal of the ACM, 33:151–178, 1986.

[23] U. Endriss. Temporal logics for representing agent communication protocols. In Frank P. Dignum, Rogier M. Eijk, and Roberto Flores, editors, Agent Communication II, pages 15–29. Springer-Verlag, 2006.

[24] P. Erdős and A. Rényi. On the evolution of random graphs. In Publication of the Mathematical Institute of the Hungarian Academy of Sciences, pages 17–61, 1960.

[25] I. Erkmen, A.M. Erkmen, F. Matsuno, R. Chatterjee, and T. Kamegawa. Snake robots to the rescue! Robotics Automation Magazine, 9(3):17–25, 2002.

[26] E. Filiot, N. Jin, and J. F. Raskin. An antichain algorithm for LTL realizability. In A. Bouajjani and O. Maler, editors, Computer Aided Verification, volume 5643 of Lecture Notes in Computer Science, pages 263–277. Springer-Verlag, 2009.

[27] C. Finucane, Gangyuan J., and H. Kress-Gazit. Ltlmop: Experimenting with language, temporal logic and robot control. In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, pages 1988–1993, 2010.

[28] V. Forejt, M.Z. Kwiatkowska, G. Norman, D. Parker, and H. Qu. Quantitative multi-objective verification for probabilistic systems. In P. Abdulla and K. Leino, editors, Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, volume 6605 of LNCS, pages 112–127. Springer-Verlag, 2011.

[29] J. Gaintzarain, P. Lucio, M. Navarro, and F. Orejas. A cut-free and invariant-free sequent calculus for PLTL. In Computer Science Logic, pages 481–495, 2007.

[30] A.S. Goldstein and E.M. Reingold. The complexity of pursuit on a graph. Theoretical Computer Science, 143:93–112, 1995.

[31] M.A. Goodrich, B.S. Morse, D. Gerhardt, J.L. Cooper, M. Quigley, J.A. Adams, and C. Humphrey. Supporting wilderness search and rescue using a camera-equipped mini UAV. Journal of Field Robotics, 25:89–110, 2008.

[32] R. Gotzheim. Temporal logic and applications—A tutorial. Computer Networks and ISDN Systems, pages 203–218, 1992.
[33] C.A.R. Hoare. An axiomatic basis for computer programming. Communications of the ACM, 12(10):576–580, 1969.

[34] G.M. Hoffmann, S.L. Wasl, and C.J. Tomlin. Distributed cooperative search using information-theoretic costs for particle filters with quadrotor applications. In Proceedings of the AIAA Guidance, Navigation, and Control Conference, pages 21–24, 2006.

[35] J.S. Jennings, G. Whelan, and W.F. Evans. Cooperative search and rescue with a team of mobile robots. 1997.

[36] B. Jonsson and Y.K. Tsay. Assumption/Guarantee specifications in linear-time temporal logic. Theoretical Computer Science, 167(1&2):47–72, 1996.

[37] G. Kantor, S. Singh, R. Peterson, D. Rus, A. Das, V. Kumar, G. Pereira, and J. Spletzer. Distributed search and rescue with robot and sensor teams. In S. Yuta, H. Asama, E. Prassler, T. Tsubouchi, and S. Thrun, editors, Field and Service Robotics, volume 24, pages 529–538. Springer-Verlag, 2006.

[38] Y. Kesten, N. Piterman, and A. Pnueli. Bridging the gap between fair simulation and trace inclusion. Inf. Comput., 200:35–61, July 2005.

[39] U. Klein and A. Pnueli. Revisiting synthesis of GR(1) specifications. In Proceedings of the 6th international conference on Hardware and Software: Verification and Testing, pages 161–181. Springer-Verlag, 2011.

[40] M. Kloetzer and C. Belta. LTL planning for groups of robots. In Proceedings of the IEEE International Conference on Networking, Sensing, and Control, pages 578–583, 2006.

[41] M. Kloetzer and C. Belta. Automatic deployment of distributed teams of robots from temporal logic motion specifications. Trans. Rob., 26:48–61, February 2010.

[42] M. Kloetzer, S. Itani, S. Birch, and C. Belta. On the need for communication in distributed implementations of LTL motion specifications. In International Conference of Robotics and Automation, pages 4451–4456. IEEE, 2010.

[43] Marius Kloetzer and Calin Belta. A fully automated framework for control of linear systems from temporal logic specifications. IEEE Transactions on Automatic Control, 53:287–297, 2008.

[44] H. Kopetz. Real-Time Systems: Design Principles for Distributed Embedded Applications. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 1997.

[45] D. Kozen. Results on the propositional $\mu$-calculus. pages 348–359. 1982.

[46] H. Kress-Gazit and G.E. Fainekos. Where’s waldo? sensor-based temporal logic motion planning. In IEEE International Conference on Robotics and Automation, pages 3116–3121, 2007.

[47] H. Kress-Gazit, G.E. Fainekos, and G.J. Pappas. Temporal-logic-based reactive mission and motion planning. IEEE Transactions on Robotics, pages 1370–1381, 2009.

[48] A. Kucera and J. Strejcek. Definable by LTL formulae with a bounded nesting depth of the next-time operator, 2004.

[49] O. Kupferman, P. Madhusudan, P.S. Thiagarajan, and M.Y. Vardi. Open systems in reactive environments: Control and synthesis. In Proceedings of the 11th International Conference on Concurrency Theory, volume 1877, pages 92–107. Springer-Verlag, 2000.

[50] L. Lamport. Proving the correctness of multiprocess programs. IEEE Transactions on Software Engineering, 3:125–143, 1977.

[51] L. Lamport. The temporal logic of actions. ACM Transactions on Programming Languages and Systems, 16:872–923, 1994.

113
[52] A.S. LaPaugh. Recontamination does not help to search a graph. *Journal of the ACM*, 40:224–245, April 1993.

[53] O. Lichtenstein, A. Pnueli, and L.D. Zuck. The glory of the past. In *Proceedings of the Conference on Logic of Programs*, pages 196–218, London, UK, 1985. Springer-Verlag.

[54] T. Linder, V. Tretyakov, S. Blumenthal, P. Molitor, D. Holz, R. Murphy, S. Tadokoro, and H. Surmann. Rescue robots at the collapse of the municipal archive of cologne city: A field report. In *IEEE International Workshop on Safety Security and Rescue Robotics*, pages 1–6, 2010.

[55] S.G. Loizou and K.J. Kyriakopoulos. Automatic synthesis of multi-agent motion tasks based on ltl specifications. In *Proceedings of the 43rd IEEE Conference on Decision and Control*, volume 1, pages 153–158, 2004.

[56] P. Madhusudan and P.S. Thiagarajan. Distributed controller synthesis for local specifications, 2001.

[57] Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 205–205, New York, NY, USA, 1987. ACM.

[58] M.J. Mataric, M. Nilsson, and K.T. Simsarin. Cooperative multi-robot box-pushing. In *IEEE International Conference on Intelligent Robots and Systems*, volume 3, pages 556–561, 1995.

[59] M. Mazo, A. Davitian, and P. Tabuada. PESSOA: A tool for embedded controller synthesis. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 566–569. Springer-Verlag, 2010.

[60] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7:417–426, 1981.

[61] B. Moszkowski. A temporal logic for multilevel reasoning about hardware. *Computer*, 18(2):10–19, 1985.

[62] R. Murphy, J. Kravitz, S. Stover, and R. Shoureshi. Mobile robots in mine rescue and recovery. *Robotics Automation Magazine, IEEE*, 16(2):91–103, 2009.

[63] R.R. Murphy. Marsupial and shape-shifting robots for urban search and rescue. *Intelligent Systems and their Applications, IEEE*, 15(2):14–19, 2000.

[64] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Lecture notes in computer science. Springer, 2002.

[65] N. Nisan, T. Roughgarden, E. Tardos, and V.V. Vazirani. *Algorithmic Game Theory*. Cambridge University Press, 2007.

[66] R.J. Nowakowski and P. Winkler. Vertex-to-vertex pursuit in a graph. *Discrete Mathematics*, 43(2–3):235–239, 1983.

[67] N. Ozay, U. Topcu, and R.M. Murray. Distributed power allocation for vehicle management systems. In *IEEE Conference on Decision and Control*, 2011.

[68] N. Ozay, U. Topcu, R.M. Murray, and T. Wongpiromsarn. Distributed synthesis of control protocols for smart camera networks. *IEEE/ACM International Conference on Cyber-Physical Systems*, 0:45–54, 2011.

[69] D. Peled and T. Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Information Processing Letters*, 63:243–246, 1997.

[70] D. Peled, T. Wilke, and P. Wolper. An algorithmic approach for checking closure properties of ω-regular languages. In *Proceedings of the 7th International Conference on Concurrency Theory*, pages 596–610. Springer-Verlag, 1995.

114
N. Piterman, A. Pnueli, and Y. Sa’ar. Synthesis of Reactive(1) designs. In 7th International Conference on Verification, Model Checking and Abstract Interpretation, volume 3855 of LNCS, pages 364–380. Springer-Verlag, jan 2006. saar.

A. Pnueli. The temporal logic of programs. Annual IEEE Symposium on Foundations of Computer Science, pages 46–57, 1977.

A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, Current trends in concurrency. Overviews and tutorials, pages 510–584. Springer-Verlag, 1986.

A. Pnueli. System specification and refinement in temporal logic. In Proceedings of Foundations of Software Technology and Theoretical Computer Science, pages 1–38. Springer-Verlag, 1995.

A. Pnueli and R. Rosner. On the synthesis of a reactive module. In Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 179–190. ACM, 1989.

A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In IEEE Symposium on Foundations of Computer Science, pages 746–757, 1990.

A. Pnueli, Y. Sa’ar, and L.D. Zuck. JTLV: A framework for developing verification algorithms. In Computer Aided Verification, pages 171–174, 2010.

A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. In Proceedings of the 8th International Conference on Computer Aided Verification, volume 1102, pages 184–195. Springer-Verlag, 1996.

A. Prior. Tense-logic and the continuity of time. Studia Logica, 13:133–148, 1962.

D.K. Richardson. Robots to the rescue! Engineering Technology, 6(4):52–54, 2011.

S.J. Rosenschein and L.P. Kaelbling. The Synthesis of Digital Machines With Provable Epistemic Properties. In Theoretical Aspects of Rationality and Knowledge, pages 83–98, 1986.

K. Rustan and M. Leino. Efficient weakest preconditions. Information Processing Letters, 93(6):281–288, 2005.

A.P. Sistla. Safety, liveness and fairness in temporal logic. In Formal Aspects of Computing, pages 495–511, 1999.

A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logics. Journal of the ACM, 32:733–749, 1985.

D. Stanley and B. Yang. Fast searching games on graphs. Journal of Combinatorial Optimization, 22(4):763–777, 2011.

S. Tadokoro, H. Kitano, T. Takahashi, I. Noda, H. Matsubara, A. Shinjoh, T. Koto, I. Takeuchi, H. Takahashi, F. Matsuno, M. Hatayama, J. Nobe, and S. Shimada. The RoboCup-Rescue project: a robotic approach to the disaster mitigation problem. In IEEE International Conference on Robotics and Automation, volume 4, pages 4089–4094, 2000.

S. Thrun, W. Burgard, and D. Fox. A probabilistic approach to concurrent mapping and localization for mobile robots. Machine Learning and Autonomous Robots, (31/5), 1998.

R. Vidal, O. Shakernia, H.J. Kim, D.H. Shim, and S. Sastry. Probabilistic pursuit-evasion games: Theory, implementation and experimental evaluation. IEEE Transactions on Robotics and Automation, 18:662–669, 2002.
[89] T. Wongpiromsarn, U. Topcu, and R. M. Murray. Formal synthesis of embedded control software: Application to vehicle management systems. *Proceedings of AIAA Infotech@Aerospace*, 2011. submitted.

[90] T. Wongpiromsarn, U. Topcu, and R.M. Murray. Receding horizon control for temporal logic specifications. In *International Conference on Hybrid Systems: Computation and Control*, pages 101–110, 2010.

[91] T. Wongpiromsarn, U. Topcu, N. Ozay, H. Xu, and R.M. Murray. TuLiP: a software toolbox for receding horizon temporal logic planning. In *Proceedings of the 14th international conference on Hybrid systems: computation and control*, pages 313–314. ACM, 2011.

[92] B. Yang and Y. Cao. Standard directed search strategies and their applications. *Journal of Combinatorial Optimization*, 17:378–399, 2009.