Towards Synthesizing Smart Contracts: Reducing ATL* Synthesis to HyperLTL Synthesis

Saarland University

Department of Computer Science

Bachelor's Thesis

submitted by

Matthias Cosler

Saarbrücken, August 2019

Supervisor:	Prof. Bernd Finkbeiner, Ph.D.
Advisor:	Christoper Hahn & Jana Hofmann
Reviewer:	Prof. Bernd Finkbeiner, Ph.D.
	Prof. Dr. Cas Cremers
Submission:	28 August, 2019

Abstract

Fair exchange protocols, also known as non-repudiation protocols, are a special class of protocols that should ensure a kind of fairness. A real-world example of this problem may be a hostage exchange as portrayed in many movies. The goal is to ensure that no party can gain an advantage by fooling the other party, e.g., by denying the receiving, canceling the exchange during the process, or by waiting too long. An application for these protocols are smart contracts, also known as the contract-signing problem. This class of protocols can be modeled and verified using game-playing techniques and Alternating Time Temporal Logic (ATL resp. ATL*), an extension of Computation Tree Logic (CTL). ATL extends CTL with the ability to quantify over strategies and capabilities of players. Designing correct protocols is hard for humans as various security issues in protocols such as the Needham-Schroeder-Protocol or Kerberos have shown. Instead of relying on the skills of humans while designing protocols, we aim at synthesizing such protocols directly from ATL* specifications. In this thesis, we present a reduction from ATL* synthesis to (Hyper-)LTL synthesis. Our technique ought to be easy to implement, as general as possible and benefit from the optimizations of existing LTL synthesis implementations, such as BoSy(Hyper). The main idea in this reduction is labeling witnesses of satisfaction for path formulas into the computation tree and using these to encode the ATL path quantification.

Acknowledgements

I would like to thank Professor Finkbeiner for giving me the opportunity to do research on this interesting topic and for letting me write my Bachelor's thesis at his chair. A huge thanks also goes to my advisors, Jana Hofmann and Christopher Hahn, who always had a free minute for my ideas or problems and who helped to improve my thesis significantly. I am also grateful for the many good discussion in the lab with my fellow students. I would like to thank my family and girlfriend for the support during writing my thesis and many thanks to my advisors and friends for proof-reading the thesis. I also would like to thank Maximilian Schwenger and Alexander Weinert for creating this thesis template and letting me use it.

Contents

Chapter 1

Introduction

Smart contracts typically describe a technical, computer- or Internet-based mechanism of establishing a contract between two or more parties. Such contracts play an important role in our information society and already are a part of our digital life as, for example, in online purchases. In the future they may replace traditional legal contracts as well. As an example for smart contracts, assume that Alice wants to buy a house from Bob. She therefore would send a message to Bob with the offer to buy his house. During further messages they may negotiate about the price, while at some point both agree on the selling, respectively purchase, of the house. The goal of a smart contract is to ensure that Alice and Bob both have evidences of their agreeing or alternatively that none of both has any evidence. This assertion is also called (fair) non-repudiation. More generally, non-repudiation enforces that the author/sender of a message, e.g., a contract, cannot dispute its authorship, while he already has the evidence of receipt from the receiver and vice versa [KR03]. For illustration of a failed fair non-repudiation, see Figure 1.1. Creating non-repudiation protocols is hard, mainly because the signing of sender and receiver may differ in time and place and because they may communicate through unreliable channels. In fact, as shown in [EY80], there is no deterministic protocol without a trusted third party, which solves the fair contract signing problem.

Creating correct protocols ensuring non-repudiation is difficult for humans, not only because it includes unreliable channels and trusted third parties or probabilities, but because creating correct protocols in general is hard for humans. A programmer or engineer has to consider a huge amount of possible attacks and a small mistake can lead to, for example, huge financial losses. One of the most prominent examples for the difficulty of designing security protocols is the the Needham-Schroeder protocol. Its flaw was not detected until 17 years later [Low95]. But also flaws in a lot of other protocols such as Kerberos (2004), Microsoft Passport (2001), French Electronic Passport (2010) and most recently Bluetooth [ATR19] display the urgency to find better designing techniques. As a consequence to these experiences, computer aided design of security protocols becomes more and more important. We consider synthesis to automatically create correct protocols from given specifications.

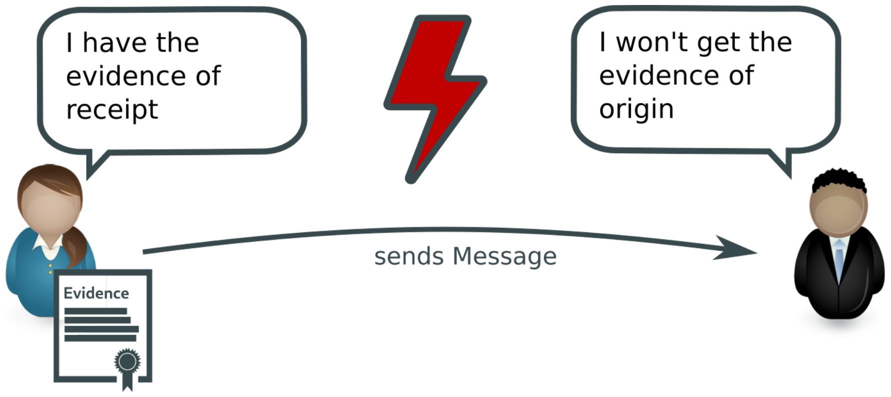

I have the evidence of receipt
I won't get the evidence of origin
Evidence
sends Message

Figure 1.1: Failed non-repudiation for Bob

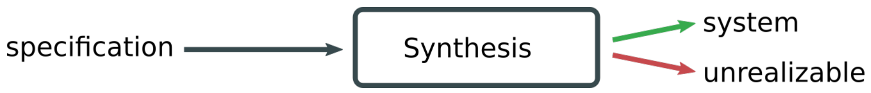


Figure 1.2: Application of synthesis

Synthesis is the process of automatically creating a system that is satisfying a given specification. Consider Figure 1.2 as an illustration. Systems are often modeled as Kripke structures, transition systems or extensions thereof, describing the behavior of the system in an optimal way. Specifications are often given in temporal logics. Temporal logics and their synthesis problem are very well studied, for example linear-time temporal logic (LTL) was introduced more than 40 years ago [Pnu77] and its synthesis problem was considered at least 20 years ago [PR89]. Moreover, specifications in temporal logics are well-structured and succinct, making it possible to be most accurate in a compact way, and therefore, have a huge advantage versus, for example, ambiguous criteria in text form. Temporal logics combine boolean operations on labels of the states of the structures with temporal operators, such as next ($\bigcirc$) or globally ($\Box$), referring to certain states. Considering, for example, the temporal logic LTL, a formula $\bigcirc\Box(a \vee b)$ holds on such computations, that, beginning in the next state, a or b holds in all following states. In Figure 1.3, we portrayed such a trace.

Alternating-time temporal logic (ATL), an extension of computation tree logic (CTL) and LTL, allows to argue about capabilities and strategies of participants. As shown in [KR03], modeling non-repudiation and fair-exchange protocols for formal verification or synthesis is possible in the alternating-time temporal logic. As an example for an ATL* formula, let $\{a_1, a_2, a_3\}$ be three participants, called agents. An ATL* formula then is

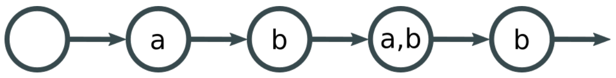


Figure 1.3: A trace where $\bigcirc\Box(a \vee b)$ holds

$\langle\langle a_1, a_2\rangle\rangle \bigcirc\Box goal$. For the formula to hold, the agents a_1 and a_2 together must have a strategy enforcing that the formula $\bigcirc\Box goal$ does hold. It, therefore, requires a system where agent a_1 together with a_2 are able to enforce paths on which, starting at the next state, each state is labeled with *goal*. Using ATL, respectively ATL*, we are not only considering the possible behavior of a system, but give participants the freedom to choose or prevent certain computations. This is perfectly suited for describing non-repudiation protocols and exchange protocols in general, i.e., protocols with no predefined order of execution and participants which are adversaries. Considering the design process of multi-agent systems using verification or synthesis, ATL is generally a good fit, as it allows to refine the restrictions of participants during the design process. At the beginning of creating a protocol, we may allow participants to make "stupid" choices resulting in the incorrectness of the protocol. At the end of the design process we should consider the most restricted case in which we do not allow participants to have the opportunity to influence the correctness of the protocol. With regard to verification, this gives us the ability to verify premature solutions. Regarding synthesis, this helps to identify incorrect specifications and also avoids unrealizable specifications that result from strict constraints.

In this thesis, we will tackle the ATL* synthesis problem by giving a reduction from ATL* synthesis to (Hyper-)LTL synthesis. HyperLTL is an extension to LTL, allowing to explicitly quantify about traces. For example *observational determinism* can be expressed in HyperLTL as follows:

$$\forall\pi\forall\pi'.\Box(I_\pi = I_{\pi'}) \rightarrow (O_\pi = O_{\pi'}),$$

with inputs I and outputs O. It states that for each pair of traces, if the inputs are equal at respectively every node, the outputs have to be equal, too.

To our knowledge, there are no existing implementations or tools for ATL or ATL* synthesis, only theoretical approaches [Sch08; SF07]. As a consequence, we aimed for an approach that is easy to implement, as general as possible and can benefit from the optimizations of existing LTL synthesis implementations such as BoSy [FFT17] or BoSyHyper [Fin+18].

Our approach uses ideas from [BSK17], which performed a similar reduction for the computation tree logic, as well as ideas from [Sch08], which first showed a 2-EXPTIME approach for ATL* synthesis. In LTL, we automatically argue about all traces of a set of traces. In ATL* however, we quantify over paths of a system. Therefore, it is absolutely necessary to transfer this possibility to LTL. Our approaches contains two main ideas. The first part is encoding ATL* features into a computation tree, which then can be easily translated to a set of traces. We will do so by adding various information to the computation tree by extending the atomic propositions. Some of these annotations are due to the fact that a system for an ATL* specification contains more information than a computation tree, some due to the fact that ATL* is strictly more expressive than LTL, even concerning the fact that we will introduce a HyperLTL formula as well. The second part is converting the ATL* formula. On the one hand we want to eliminate all

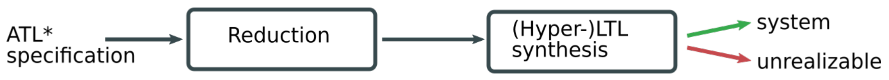


Figure 1.4: ATL* Synthesis via (Hyper-)LTL synthesis

ATL* fractions from the formula, on the other hand we want the synthesizer to solve an ATL* synthesis problem. This apparent contradiction is solved by using the encoded propositions. The encoded propositions are output variables, therefore, their values are set by the synthesis algorithm. However, we will add various LTL constraints to the LTL formula, such that we force the synthesizer to solve the ATL* synthesis problem by finding the correct setting for the new output variables. The output variables will then reveal, together with the constructed system, the ATL* synthesis solution. Figure 1.4 visualizes how ATL* synthesis is reduced to HyperLTL synthesis.

Related Work

The topic of ATL synthesis did not get as much attention as the synthesis for other logics, such as LTL. However, two approaches are known to us. The first approach was by Schewe and Finkbeiner [SF07], considering the distributed synthesis for ATL. While this is an interesting problem since ATL is well suited for describing distributed systems we were interested in the non-distributed synthesis setting. This is due to the fact that distributed synthesis in general is very hard and becomes undecidable even in small settings. As the authors showed, distributed synthesis for ATL and even CTL becomes undecidable in any non-hierarchical setting. In 2008, Schewe [Sch08] showed that the synthesis and realizability for ATL* is 2-EXPTIME complete, which was a major step, as up to then, ATL* synthesis was solved with a detour over the μ-calculus, generating an additional exponential blow-up. The author, however, presented an approach for solving the synthesis problem for ATL* without this detour. He gave a transformation from concurrent game structures, the common models for ATL, to special trees, which then got parsed by a customized tree automaton. We adapted some of these findings in our approach.

Smart contracts became a well-known topic in the last years, most notably in the context of block chains. The most common properties in the context of smart contracts as protocols are the contract-signing problem, first introduced in [Blu83], fair exchange and non-repudiation [PVG03; KMZ02; KR03] and abuse freeness [GJM99; KR02].

Chatterjee and Raman [CR12] described an approach to the synthesis of exchange protocols, using LTL as specification language. They found out that strictly competitive formulations are too strong for synthesizing such protocols and therefore switched to conditionally competitive co-synthesis, similar to assume-guarantee synthesis [CA07].

Outline

In Chapter 2, we give an introduction to the topics we will use in our approach. We will go into the topics of synthesis, LTL, HyperLTL as well as ATL, respectively, ATL*. In Chapter 3, we reduce ATL* synthesis to (Hyper-)LTL synthesis, based on concurrent game structures. In Chapter 4, we present a second approach for reducing ATL* synthesis to (Hyper-)LTL synthesis, based on alternating transition systems. In Chapter 5, we analyze protocols for smart contracts in the context of ATL synthesis and give an outlook for future work on this topic. In Chapter 6, we shortly summarize our approaches to compare and discuss them. We will give an outlook on future work considering our approaches on reducing ATL synthesis to (Hyper-)LTL synthesis.

Chapter 2

Preliminaries

In this chapter we lay the foundation for understanding the following reductions. We will introduce the problem of synthesis and realizability, several structures for ATL* and LTL verification / synthesis, such as trace models, computation trees, concurrent game structures and alternating transition systems. Further we will introduce syntax and semantics of linear-time temporal logic (LTL) and alternating-time temporal logic (ATL / ATL*).

2.1 Synthesis

Synthesis is the problem of finding a correct implementation to a specification. We consider temporal specifications, more precisely specifications in the linear-time temporal logic (LTL) and alternating-time temporal logic (ATL resp. ATL*) over a set of atomic propositions (AP). Propositions are partitioned into inputs I and outputs O : $I \dot\cup O = AP$. Given a specification, a synthesis algorithm outputs either a system or the term unrealizable. The system has the property of modeling or realizing the specification, hence the specification has to hold for all possible infinite sequences of input sets $(2^I)^\omega$. In each step of a computation, the inputs are chosen by the environment, whereas the outputs are chosen by the system in such a way, that the specification holds. A specification is called *realizable*, if there exists a system choosing outputs, such that the specification holds for all possible inputs. Analogously a system is *unrealizable* if there is no succeeding system.

2.2 Computation Trees

An infinite labeled tree is a tuple (D, L, V, l), where

- D is a set of directions.
- L is a set of labels.
- $V := D^*$ is a set of nodes. We expect the following constraints on V to hold:

   1. $\varepsilon \in V$ is the root.
   2. $\forall n \in V . \exists d \in D .\ n \cdot d \in V$, hence there are no leafs.
- l is a labeling function: $l : V \to 2^L$, which annotates each node with a combination of labels.

We call a tree a computation tree with inputs I and outputs O, iff it is a tuple $(D = 2^I, L = 2^O, V, l)$. We define a path through such a tree as π, which can be infinite: $\pi \in D^\omega$, or finite: $\pi \in D^*$. Let us remark that based on this definition, nodes and finite paths are equivalent.

2.3 Linear-time Temporal Logic (LTL)

LTL [Pnu77] combines boolean operations with temporal operators as *next* ($\bigcirc$) and *eventually* ($\diamondsuit$). LTL is evaluated over trace models of atomic propositions. Boolean operations are evaluated over the propositions of states, whereas temporal operators refer to the states on which the propositions are evaluated. As an example, *next* refers to the state which comes relatively next to the current state and *eventually* refers to some state coming in a finite future, hence a state which comes eventually or finally.

2.3.1 LTL Syntax

$$\varphi \coloneqq p \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg \varphi \mid \bigcirc \varphi \mid \varphi \,\mathcal{U}\, \varphi$$

where p is an atomic proposition: $p \in AP$, which can be parted into inputs I and outputs O: $AP = I \dot\cup O$.

2.3.2 LTL Semantics

As before let $AP = I \dot\cup O$ the set of atomic propositions, consisting of inputs I and outputs O. Generally LTL is defined over a set of traces: $TR \coloneqq (2^{AP})^\omega$.

Let $\pi \in TR$ be trace, $\pi_{[0]}$ the starting element of a trace π and for a $k \in \mathbb{N}$ and be $\pi_{[k]}$ be the k-th element of the trace π. With $\pi_{[k,\infty]}$ we denote the infinite suffix of π starting at k. We write $\pi \models \varphi$ for *the trace* π *satisfies the formula* φ.
For a trace $\pi \in TR$, $p \in AP$ and path formulas ψ:
- $\pi \models \neg\psi$ iff $\pi \not\models \psi$
- $\pi \models p$ iff $p \in \pi_{[0]}$; $\pi \models \neg p$ iff $p \notin \pi_{[0]}$.
- $\pi \models \psi_1 \wedge \psi_2$ iff $\pi \models \psi_1$ and $\pi \models \psi_2$. Similarly for $\psi_1 \vee \psi_2$
- $\pi \models \bigcirc \psi$ iff $\pi_{[1]} \models \psi$
- $\pi \models \psi_1 \,\mathcal{U}\, \psi_2$ iff $\exists l \in \mathbb{N} : (\pi_{[l,\infty]} \models \psi_2 \wedge \forall m \in [0, l-1] : \pi_{[m,\infty]} \models \psi_1)$

Further temporal operators as *eventually* ($\diamondsuit$), *release* ($\mathcal{R}$) or *globally* ($\square$) can be deduced from the defined operators as usual.

In the case of deterministic synthesis, where each sequence of outputs belongs to a unique sequence of inputs, we assume w.l.o.g., that each set of traces can be transformed into a computation tree and each computation tree can be transformed into a set of traces. Since for our purposes, the similarity to the definition of ATL* semantics is highly important, we write $TR \models \varphi$ for *all traces in the trace set TR satisfy the formula* φ as well as $C \models \varphi$ for *all paths of the computation tree C satisfy the formula φ.*

2.4 HyperLTL

The definition of HyperLTL is taken from [Coe+19].

HyperLTL [Cla+14] extends LTL (Section 2.3) with explicit trace quantification. Let $\mathcal{V} = \{\pi_1, \pi_2, \cdots\}$ be an infinite set of trace variables. HyperLTL formulas are defined by the grammar:

$$\varphi \coloneqq \forall \pi.\, \varphi \mid \exists \pi.\, \varphi \mid \psi$$

$$\psi \coloneqq a_\pi \mid \neg\psi \mid \psi \vee \psi \mid \bigcirc \psi \mid \psi \,\mathcal{U}\, \psi,$$

where $a \in AP$ and $\pi \in \mathcal{V}$. Here, $\forall \pi.\, \varphi$ and $\exists \pi.\, \varphi$ denote universal and existential trace quantification, and a_π requires the atomic proposition a to hold on trace π. The semantics of HyperLTL is defined with respect to a set of traces T. Let $\Pi : \mathcal{V} \mapsto T$ be a trace assignment that maps trace variables to traces in T. We can update a trace assignment Π, denoted by $\Pi[\pi \mapsto t]$, where π maps to t and all other trace variables are as in Π. The satisfaction relation $\models_T$ for HyperLTL over a set of traces T is defined as follows:

$$\begin{array}{lll}
\Pi, i \models_T a_\pi & \text{iff} & a \in \Pi(\pi)[i] \\
\Pi, i \models_T \neg\varphi & \text{iff} & \Pi, i \not\models_T \varphi \\
\Pi, i \models_T \varphi_1 \vee \varphi_2 & \text{iff} & \Pi, i \models_T \varphi_1 \text{ or } \Pi, i \models_T \varphi_2 \\
\Pi, i \models_T \bigcirc\varphi & \text{iff} & \Pi, i+1 \models_T \varphi \\
\Pi, i \models_T \varphi_1 \,\mathcal{U}\, \varphi_2 & \text{iff} & \exists j \geq i.\ \Pi, j \models_T \varphi_2 \\
 & & \qquad \wedge\, \forall i \leq k < j.\ \Pi, k \models_T \varphi_1 \\
\Pi, i \models_T \exists \pi.\, \varphi & \text{iff} & \exists t \in T.\ \Pi[\pi \mapsto t], i \models_T \varphi \\
\Pi, i \models_T \forall \pi.\, \varphi & \text{iff} & \forall t \in T.\ \Pi[\pi \mapsto t], i \models_T \varphi.
\end{array}$$

We say that a trace set T satisfies a HyperLTL formula φ, written as $T \models \varphi$, if $\emptyset, 0 \models_T \varphi$, where $\emptyset$ denotes the empty trace assignment.

For this work, we are interested in the property of *observational determinism.* The following HyperLTL formula states observational determinism: $\forall\pi\forall\pi' \square(I_\pi = I_{\pi'}) \rightarrow (O_\pi = O_{\pi'})$, with inputs I and outputs O. It states that for each pair of traces, if the inputs are equal at respectively every node, the outputs have to be equal to. We will use a slightly modified version of this formula in our approaches. As shown in [Fin+18], synthesizing such HyperLTL formulas with a bounded approach gives a quadratic blowup in relation to LTL synthesis.

Similar to LTL, we assume w.l.o.g. that, sets of sets of traces can be transformed to sets of computation trees and backwards.

2.5 Alternating-time Temporal Logic (ATL)

ATL is a branching time logic such as CTL [EH86] but with finer-grained path quantification. Similar to LTL it combines boolean operations over propositions with temporal operators, but as an extension to LTL, also path quantification. ATL is evaluated over tree-like structures. In branching time logics, specifications can be written, that enforce different properties on different branches of a system. That is in contrary to LTL, where a property has to hold generally on all possible traces. In ATL branches can be selected through path quantifiers, which quantify over capabilities and strategies of agents. As a strategy we denote the scheme of an agent to enforce properties. In other words, an agent has a strategy, if he is able to enforce these properties.

Therefore, in comparison to computation tree models and trace models, we need models containing behavior and capabilities of each agent. ATL was originally defined on alternating transition systems (ATS) [AHK98], whereas later on, the authors redefined the logic over concurrent game structures (CGS) [AHK02]. We will introduce both definitions and base our reduction on both approaches.

ATL can be extended to ATL*, which allows to express temporal modalities on paths without simultaneously quantifying paths. ATL* therefore extends both, LTL and ATL and is strictly more expressive than both logics. For the content of this work, we will only consider ATL*.

Example 2.1 (ATL*)**.** As an example for an ATL* specification, let us consider the following formula. For three agents $\{a_1, a_2, a_3\}$, the following formula has to hold:

$$\langle\langle a_1, a_2 \rangle\rangle \bigcirc \Box \mathit{goal}$$

As mentioned above, $\bigcirc$ relates to the next state, whereas $\Box$ relates to all states. With $\langle\langle a_1, a_2 \rangle\rangle$ we quantify paths, hence select paths on which agents a_1 and a_2 have a strategy of enforcing the formula $\bigcirc \Box \mathit{goal}$. Equally spoken, for the whole formula to hold, the agent a_3 must not have a counter strategy, enforcing that the formula $\bigcirc \Box \mathit{goal}$ does not hold. On all of the paths, the path quantifier chooses, the subformula $\bigcirc \Box \mathit{goal}$ has to hold. Let us analyze each of the paths individually. For the state where the formula is evaluated we require no boolean constraints, but as described with $\bigcirc$, at the next state $\Box \mathit{goal}$ has to hold. A path satisfying this subformula therefore has to have the proposition goal in each state of the path starting at the state where $\Box$ is evaluated. Summarizing the formula requires a system where agent a_1 together with a_2 are able to enforce paths on which, starting at the next state, each state is labeled with goal.

$\triangle$

2.5.1 Concurrent Game Structures

In this subsection, we look at concurrent game structures (CGS). While in Kripke structures, a state transition represents a step of a closed system, in a concurrent game structure, a state transition corresponds to a possible step in a game between the agents that constitute the system. Each agent in each state has the choice of doing several moves, which will lead to different evaluations of the system. Moves of all agents (one move per agent) are combined to move vectors, which will directly influence the next state. The following definition is based on [AHK02].

Definition 1 (Concurrent Game Structures)**.** A concurrent game structure is a tuple $S = (A, Q, q_0, AP, l, \vec{M}, \tau)$ with the following components:

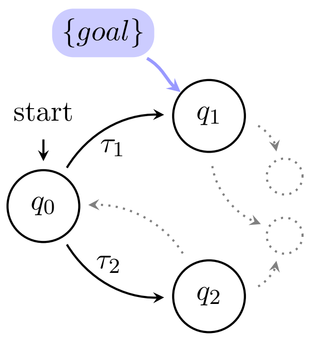
- $A = \{a_1, \cdots, a_k\}$, a set of *agents* or players of size k.
- Q, a finite set of *states*, with q_0 the *initial state.*
- AP, a finite set of *atomic propositions.*
- $l : Q \to 2^{AP}$, the *labeling function*, which determines a set of atomic propositions which hold in each state.
- $\vec{M}$, a set of possible *move vectors.* A move vector $\vec{m}$ is an aggregation of moves m, where each move belongs to an agent. For each agent $a \in \{a_1, \cdots, a_k\}$ and each state $q \in Q$, we denote the set of moves available as M_a^q. As an upper bound on possible moves per agent, regardless of the state, we introduce the notation M_a. For each state $q \in Q$, the move vector $\vec{m}$ at q therefore is a vector $\langle m_1, \cdots, m_k \rangle$ such that $m_i \in M_{a_i}^q$ for each player $a_i \in A$. We define the set $\vec{M}^q$ as the set of all move vectors available at a state q.
- τ, the *transition function.* For each state $q \in Q$ and each move vector $\langle m_1, \cdots, m_k \rangle$ where $m_i \in M_{a_i}^q$ for each player a_i, the transition function determines a state $\tau(q, \langle m_1, \cdots, m_k \rangle) \in Q$ that results from state q, if every player $a_i \in A$ chooses move m_i.

Example 2.2 (Running Example For Concurrent Game Structures)**.** We provide a running example for concurrent game structures. Let the agents be $A = \{a_1, a_2\}$, the moves per agent and state: $M_{a_1}^{q_0} = \{m_1, m_1'\}$ and $M_{a_2}^{q_0} = \{m_2\}$. We only outline the behavior of the system for the first transition. All further transitions and states are just for illustration. The transition function for state q_0, therefore is $\tau(q_0, \langle m_1, m_2 \rangle) = q_1$ and $\tau(q_0, \langle m_1', m_2 \rangle) = q_2$. We label the state q_o with *goal*. A sketch of the system is provided in Figure 2.1.

$\triangle$

2.5.2 Alternating Transition Systems

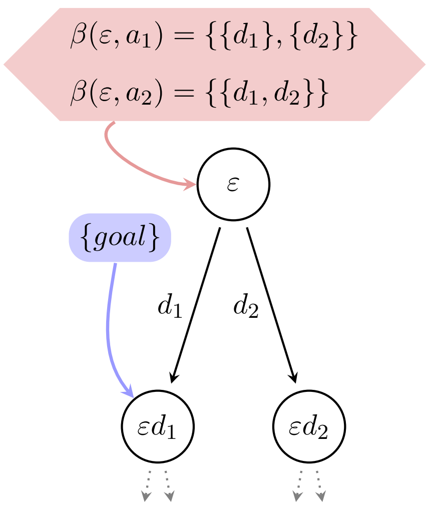
We evaluate ATL* formulas over alternating transition systems. While in ordinary transition systems, each transition corresponds to a possible step of the system, in alternating transition systems, each transition corresponds to a possible step in a game between the agents that constitute the system. Each agent in each state has the choice of doing



$$\tau_1 \coloneqq \tau(q_0, \langle m_1, m_2 \rangle)$$

$$\tau_2 \coloneqq \tau(q_0, \langle m_1', m_2 \rangle)$$

Figure 2.1: Example of a CGS


$\beta(\varepsilon, a_1) = \{\{d_1\}, \{d_2\}\}$
$\beta(\varepsilon, a_2) = \{\{d_1, d_2\}\}$
ε
$\{goal\}$
d_1
d_2
εd_1
εd_2

$$d_1 = \emptyset$$

$$d_2 = \{i_1\}$$

Figure 2.2: Example of an ATS tree

several moves, which will lead to different evaluations of the system. Moves are sets of possible next states, therefore agents can restrict the common possible next states by choosing moves until, in the case of all agents choosing a move, only one possible next state is left. We obtained the following definition from [AHK98].

In the following we often build the power set of directions or states ($2^Q, 2^{2^Q}$ respectively $2^D 2^{2^D}$). For these power sets we always ignore the empty set $\emptyset$. For example for $Q = \{q_1, q_2\}$, the powerset is $2^Q = \{\{q_1\}, \{q_2\}, \{q_1, q_2\}\}$ and $2^{2^Q} = \{\{\{q_1\}\}, \{\{q_2\}\}, \{\{q_1, q_2\}\}, \{\{q_1\}, \{q_2\}\}, \{\{q_1\}, \{q_1, g_2\}\}, \{\{q_2\}, \{q_1, g_2\}\}\}$.

Definition 2. An alternating transition system (ATS, for short) is a 5-tuple (AP, Q, l, A, τ) with the following components:

- AP is a set of *propositions*.
- Q is a set of *states*, q_0 is the *initial state*.
- $l : Q \to 2^{AP}$, a *labeling function*.
- A is a set of *agents*.
- $\tau : Q \times A \to 2^{2^Q}$ is a *transition function* that maps a state and an agent to a nonempty set of moves, where each move is a set of possible next states. For an agent a and a state q, we denote a move: $M_a^q \in \tau(q, a)$. Possible next states are limited by the set of moves, therefore the set of possible next states per agent, we call them capabilities, can be defined as $\mathcal{C}_a^q \coloneqq \bigcup_{M_a^q \in \tau(q,a)} M_a^q$, the union of all his possible moves. Consequentially for a fixed agent a, a move is a subset of all his capabilities, which is of course a subset of all generally possible next states ($M_a^q \subseteq \mathcal{C}_a^q \subseteq Q$). We require that the intersection of all possible moves for all

agents is a singleton: For $A = \{a_1, \cdots, a_m\}$, every state $q \in Q$ and one arbitrary move for each agent: $M^q_{a_i} \in \tau(q, a_i)$, the intersection $M^q_{a_1} \cap \cdots \cap M^q_{a_m}$ is a singleton.

Trees of Alternating Transition Systems

Trees of alternating transition systems (ATS trees) are an extension of trees that handle multiple agent systems and their possible behavior. We also show that every alternating transition system has an unrolling which is an ATS tree.

Definition 3. An ATS tree is a 6-tuple (D, L, V, l, A, β), where

- D, L, V, l represent a general *tree* as defined in Section 2.2.
- A will be a set of *agents* or actors which have a possible behavior.
- β describes the possible *behavior* of agents: $\beta : V \times A \to 2^{2^D}$. The behavior function is constructed using the transition function from Definition 2. Instead of states we talk about nodes as a history of states. Further we map to directions instead of states, because directions are sufficient to determine the next node. For an agent a and a node n, we denote a move: $M^n_a \in \beta(n, a)$. Capabilities are defined as $\mathcal{C}^n_a := \bigcup_{M^n_a \in \beta(n,a)} M^n_a$ and we require that the intersection of all possible moves for all agents is a singleton: For $A = \{a_1, \cdots, a_m\}$, every node $n \in V$ and one arbitrary move for each agent: $M^n_{a_i} \in \beta(n, a_i)$, the intersection $M^n_{a_1} \cap \cdots \cap M^n_{a_m}$ is a singleton.

For the purpose of synthesis we expect an ATS tree to extend a computation tree, hence it is the tuple $T = (D = 2^I, L = 2^O, V, l, A, \beta)$

Example 2.3 (Running Example For Trees of Alternating Transition Systems)**.** We provide a running example for an ATS tree. Let the agents be $A = \{a_1, a_2\}$, the behavior per agent and state: $\beta(\varepsilon, a_1) = \{\{d_1\}, \{d_2\}\}$ and $\beta(\varepsilon, a_2) = \{\{d_1, d_2\}\}$. We only outline the behavior of the system for the first transition. All further transitions and states are just for illustration. We label the node εd_1 with *goal*. A sketch of the system is provided in Figure 2.2.

We may examine if the single requirement is satisfied:

$$\{d_1\} \cap \{d_1, d_2\} = \{d_1\}$$

$$\{d_2\} \cap \{d_1, d_2\} = \{d_2\}$$

Additionally $\mathcal{C}^{a_1}_\varepsilon = \{d_1, d_2\}$ and $\mathcal{C}^{a_2}_\varepsilon = \{d_1, d_2\}$

△

Unrolling of Alternating Transitions Systems

We present an unrolling of alternating transition systems to ATS trees.

Let us first define that the set AP of propositions can be parted into inputs I and outputs O: $I \dot\cup O = AP$. We then split the labeling function l into $l_i : Q \to 2^I$ and $l_o : Q \to 2^O$, such that for a state q: $l(q) = l_i(q) \dot\cup l_o(q)$. In the case of deterministic synthesis, all successors of a state have a distinct combination of inputs, respectively there are no two successors q' and q'' of an arbitrary state q, such that $l_i(q')$ and $l_i(q'')$ are equal. Further we call the power set of inputs $2^I = D$ directions, and a node a history of directions and the root node is called ε.

Let $u : Q \to V$ be the unrolling of states to nodes:

$$\begin{aligned} \varepsilon &= u(q_0) \\ n \cdot d &= u(q_i) \text{ where } \qquad l_i(q_i) \in D \ \wedge \ \bigwedge_{a \in A} q_i \in \mathcal{C}_a^q \ \wedge \ n = u(q_{i-1}) \end{aligned}$$

Based on this definition, a state q_i unrolls to a node, which is either the root node or a node $n \cdot d$ if

- the direction d leading to this node is the set of inputs $l_i(q)$ and
- the state q_i is a possible successor of a state q_{i-1} for all agents and
- the predecessor node n is the unrolling of the predecessor state q_{i-1}

Additionally let $u_{set} : 2^{2^Q} \to 2^{2^V}$ be the unrolling of elements from 2^{2^Q}.

$$u_{set}(S) = \{\{u(q) \mid q \in S'\} \mid S' \in S\}$$

The ATS tree (D, L, V, l, A, β), based on the unrolling, is a tree such that:

- D, L, V are a general computation tree as defined in Section 2.2
- $l(n) = l_o(q_i) \wedge u(q_i) = n$ is a labeling function, which annotates each node with its state output variables.
- A is a set of agents as defined in Definition 2.
- $\beta(n, a) = u_{set}(\tau(q_i, a)) \wedge u(q_i) = n$ is the behavior function as an exact unrolling of the transition function from Definition 2.

2.5.3 ATL* Syntax

$$\begin{aligned} \varphi &:= true \mid false \mid o \mid \neg o \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \langle\langle A' \rangle\rangle \psi \\ \psi &:= \varphi \mid i \mid \neg i \mid \psi \wedge \psi \mid \psi \vee \psi \mid \bigcirc \psi \mid \psi \,\mathcal{U}\, \psi \mid \psi \,\mathcal{R}\, \psi \end{aligned}$$

where $i \in I$ are inputs and $o \in O$ are Outputs. An ATL formula $\langle\langle A' \rangle\rangle \psi$ is called basic. For a basic formula we call the set of agents $A' \subseteq A$ a coalition and the formula ψ the body of a basic formula.
In the following we define the semantics of ATL*. First based on concurrent game structures, secondly based on trees of alternating transition systems.

2.5.4 ATL* Semantics over Concurrent Game Structures

First, we consider the Definition of ATL* Semantics over concurrent game structures (➔Definition 1). The semantics definition of ATL* is based on [AHK02].

➔Sec. 2.5, Page 11

For path formulas, boolean operations and proposition evaluation the ATL* semantics is similar to LTL Section 2.3), [Pnu77] or CTL* [EH86]. Basic formulas however, are interpreted using the following game. Assume that we evaluate some basic formula over a system S at a state q. The game has two parties, one party operates all agents from the coalition of the basic formula, we call this the protagonist. The antagonist however chooses the next transition but has to comply with the choices the protagonist made. First the protagonist chooses moves on behalf of the agents in the coalition. Each agent in the coalition chooses exactly one move. Secondly the antagonist chooses the next transition, but he has to follow these rules:

1. The transition has to be possible for all agents, which are not in the coalition.
2. The direction has to comply with the moves, the coalition, hence protagonist, chose.

That way the game generates an infinite path on which the body of the basic formula needs to hold. In fact, since the antagonist can not be controlled, we get a set of infinite paths based on the possible choices of the antagonist. The body of the basic formula needs to hold on all of these infinite paths.

Formally, for a concurrent game structure we define a strategy for an agent $a \in A$ as a function that maps every nonempty finite state sequence to a move $f_a : Q^+ \to M_a^q$, where for each sequence $q_0 \cdot q_1 \cdots q_i = \lambda \in Q^+$, the last state is $q = q_i$. For a set of agents $A' \subseteq A$, hence a coalition, we call a set of strategies:

$$F_{A'} := \{f_a \mid a \in A'\}.$$

We call an infinite path through a CGS $\pi : Q^\omega$ and call it a *play* if it is the outcome of a strategy. Based on this, the set of plays beginning in some state q, which can be enforced using a set of strategies $F_{A'}$ is defined as

$$plays(q, F_{A'}) = \{q_0 \cdot q_1 \cdots \mid \forall i \geq 1 . q_i = \tau(q_i, \langle m_1, \cdots, m_k \rangle) \wedge \forall a_j \in A' . m_j = f_a(q_0 \cdots q_i)\}.$$

Therefore a path is in the set of $plays(q, F_{A'})$, if all directions of this path are the result from transitions, where the move vector matches the strategies of the coalition. Because the transition function is a function, we can guarantee that for any coalition A', any strategy $F_{A'}$ and any state q, there is always at least one path in the set $plays(n, F_{A'})$

Let S be a CGS. We write $S, q \models \varphi$ for *the state satisfies formula φ in the CGS S*. If S is clear from the context we simply write $q \models \varphi$. Instead of $S, \varepsilon \models \varphi$ we sometimes shortly write $S \models \varphi$.

Let $\pi = q_0 \cdot q_1 \cdots \in Q^\omega$ be an infinite path through the CGS, beginning in the state q. For a $k \in \mathbb{N}$, let $\pi_{[k]}$ be the k-th state of the path π. Consequently let $\pi_{[0]} = q_0$ and $\pi_{[k,\infty]}$ be the infinite suffix of π starting at the k-th state after q_0: $\pi_{[k]} \cdot \pi_{[k+1]} \cdots$. We write $\pi \models \varphi$ for *the path π satisfies formula φ*.

For states $q \in Q$, outputs $o \in O$, state formulas φ and path formulas ψ:

- $q \models \neg\varphi$ iff $n \not\models \varphi$
- $q \models o$ iff $o \in l(q)$
- $q \models \varphi_1 \wedge \varphi_2$ iff $q \models \varphi_1$ and $q \models \varphi_1$. Similarly for $\varphi_1 \vee \varphi_2$
- $q \models \langle\langle A' \rangle\rangle \psi$ iff $\exists F_{A'} . \forall \pi \in plays(q, F_{A'}) .\ \pi \models \psi$

For an infinite path $\pi = q_0 \cdot q_1 \cdots \in Q^\omega$ and inputs $i \in I$:

- $\pi \models \varphi$ iff $\pi_{[0]} \models \varphi$

- $\pi \models i$ iff $i \in \pi_{[1]}$; $\pi \models \neg i$ iff $i \notin \pi_{[1]}$.
- $\pi \models \psi_1 \wedge \psi_2$ iff $\pi \models \psi_1$ and $\pi \models \psi_2$. Similarly for $\psi_1 \vee \psi_2$
- $\pi \models \bigcirc \psi$ iff $\pi_{[1,\infty]} \models \psi$
- $\pi \models \psi_1 \mathcal{U} \psi_2$ iff $\exists l \in \mathbb{N} : (\pi_{[l,\infty]} \models \psi_2 \wedge \forall m \in [1, l-1] : \pi_{[m,\infty]} \models \psi_1)$
- $\pi \models \psi_1 \mathcal{R} \psi_2$ iff $(\forall l \in \mathbb{N} : \pi_{[l,\infty]} \models \psi_2) \vee (\exists l \in \mathbb{N} : \pi_{[l,\infty]} \models \psi_1 \wedge \forall m \in [1, l] : \pi_{[m,\infty]} \models \psi_2)$

The semantics definition is very similar to semantics definition of other temporal logics as CTL* and LTL. In contrast to LTL or CTL*, we quantify over strategies of agents, hence a path formula ψ is satisfied iff all *plays* starting in q satisfy ψ. Also note that input satisfaction is shifted, in other words inputs are satisfied if the path reaches the next node with this input, not if the input was chosen to reach the current node.

2.5.5 ATL* Semantics over Alternating Transition-system Trees

➔Sec. 2.5, Page 13

Since trees are a better fit to the problem of synthesis, we define the semantics of ATL* over trees, especially trees of alternating transition systems (➔Definition 3) instead of alternating transition systems. As a tree of an alternating transition system (ATS tree) is the unraveling of an ATS, the definition also follows [AHK98]. The intuition of considering a basic formula as a game between the agent is analogue to Subsection 2.5.4. Formally, for an ATS tree we define a strategy for an agent $a \in A$ as $f_a : V \to 2^D$. For each given node n, hence history of directions, the strategy $f_a(n)$ chooses a move $M_a^n \in \beta(n, a)$. The strategy for player a therefore induces plays that player a can enforce. For a set of agents $A' \subseteq A$, hence a coalition, we call a set of strategies:

$$F_{A'} := \{f_a \mid a \in A'\}.$$

The set of capabilities of a set of agents $A' \subseteq A$ is defined as

$$\mathcal{C}_n^{A'} = \bigcap_{a \in A'} \mathcal{C}_n^a.$$

We define an infinite path through such a tree as $\pi : D^\omega$ and call it a *play* if it is the outcome of a set of strategies $F_{A'}$. Based on this, the set of plays beginning in some node n, which can be enforced using a strategy $F_{A'}$ is defined as

$$plays(n, F_{A'}) = \{nd_1 \cdots \mid \forall i \geq 1.d_i \in \mathcal{C}_{n \cdots d_{i-1}}^{A \setminus A'} \wedge \forall f_a \in F_{A'}.d_i \in f_a(n \cdots d_{i-1}).$$

Accordingly to this formula, a path is in the set of $plays(n, F_{A'})$, if all directions of this path are in the strategy of the coalition and all agents not in the coalition, are capable of following in the same direction. As a result of the singleton requirement, we can guarantee that for any coalition A', any strategy $F_{A'}$ and any node n, there is always at least one path in the set $plays(n, F_{A'})$.

Let S be an ATS tree. We write $S, n \models \varphi$ for *the node satisfies formula φ in the tree S*. If S is clear from the context we simply write $n \models \varphi$. Instead of $S, \varepsilon \models \varphi$ we sometimes shortly write $S \models \varphi$.

Let $\pi \in V^\omega$ be an infinite path through the ATS tree, starting in the node n. For a $k \in \mathbb{N}$ and i directions leading to node n , let $\pi_{[k]}$ be the $(i+k)$-th direction of the path π, respectively the k-th direction after n. Consequently let $\pi_{[0]} = n$ and $\pi_{[k,\infty]}$ be the infinite suffix of π starting at the k-th direction after n: $\pi_{[k]} \cdot \pi_{[k+1]} \cdots$. We write $\pi \models \varphi$ for *the path π satisfies formula φ*.

The following satisfaction relation is very similar to the satisfaction relation using alternating transition systems (Subsection 2.5.4). Instead of states we consider nodes and of course we use the newly defined formula describing *plays*.

For nodes $n \in V$, outputs $o \in O$, state formulas φ and path formula ψ:

- $n \models \neg\varphi$ iff $n \not\models \varphi$
- $n \models o$ iff $o \in l(n)$
- $n \models \varphi_1 \wedge \varphi_2$ iff $n \models \varphi_1$ and $n \models \varphi_1$. Similarly for $\varphi_1 \vee \varphi_2$
- $n \models \langle\langle A' \rangle\rangle \psi$ iff $\exists F_{A'}.\forall \pi \in plays(n, F_{A'}).\ \pi \models \psi$

For a path $\pi \in V^\omega$:

- $\pi \models \varphi$ iff $\pi_{[0]} \models \varphi$
- $\pi \models i$ iff $i \in \pi_{[1]}$; $\pi \models \neg i$ iff $i \notin \pi_{[1]}$.
- $\pi \models \psi_1 \wedge \psi_2$ iff $\pi \models \psi_1$ and $\pi \models \psi_2$. Similarly for $\psi_1 \vee \psi_2$
- $\pi \models \bigcirc\psi$ iff $\pi_{[1,\infty]} \models \psi$
- $\pi \models \psi_1 \mathcal{U} \psi_2$ iff $\exists l \in \mathbb{N} : (\pi_{[l,\infty]} \models \psi_2 \wedge \forall m \in [1, l-1] : \pi_{[m,\infty]} \models \psi_1)$

2.5.6 Alternating Transition Systems vs. Concurrent Game Structures

In this section we analyze the differences between alternating transition systems and concurrent game structures. Both are structures used to construct models for ATL* specifications and can be used to evaluate ATL* specifications. Alternating transition systems are an extension of transition systems with alternating branching semantics. In alternating transition systems, the choices of players influence the behavior of the system in the following way. A coalition of agents determine a set of possible next states, by choosing a set of states for each agent, which then get intersected. Assuming that all agents choose a set of possible next states the successor state is determined, if not, all possible successor states have to be considered equally. Concurrent game structures however extend Kripke structures, such that they allow choices of players, that influence the behavior of the system similar to alternating transition systems. in comparison to alternating transition systems, in concurrent game structures the authors extracted the alternating characteristic from the transition function into move vectors. Instead of choosing sets of possible successor states, the agents choose moves, which get combined to move vectors. In the transition function, each move vector may project to a different successor state. In the case of multiple possible move vectors, hence not all agents choosing moves, all transitions having one of these move vectors and consequently all states consulting these transitions, have to be considered equally.

Concurrent Game Structures are a better fit to the synthesis problem, since they can be trees, as we will show in ➔Subsection 3.1.2. Alternating transition systems can not be trees, which is the reason, that we defined ATS trees as the unrolling of alternating transition systems.

➔Sec. 3.1, Page 20

Chapter 3

Reduction from Concurrent Game Structures

The approach in this chapter is based on concurrent game structures. We show how concurrent game structures are transformed into computation trees, while additional propositions and (Hyper-)LTL formulas guarantee the correctness of of the computation tree. Finally, we replace parts of the ATL* formula such that we obtain a pure LTL formula that depicts, in combination with the added LTL formulas and a small HyperLTL formula, the new (Hyper-)LTL specification. This LTL specification is realizable if and only if the ATL* specification is realizable.

We structure this chapter into two parts. In the first part, we transform concurrent game structures into so called explicit models. We call a concurrent game structure an explicit model of a specification if the model indicates its satisfaction, more precise, the satisfaction of its path formulas. This part is mostly based on [Sch08]. In the second part we adapt the findings of the first part to our goal, finding a reduction from ATL* synthesis to (Hyper-)LTL synthesis. Mostly, we explain which propositions and formulas are necessary, to transform explicit models to regular computation trees.

3.1 From Models to Explicit Models

In [Sch08] it is shown that the synthesis of ATL* is 2EXPTIME-complete by constructing an automaton, which is testing only a particular set of models. This particular set of models is called *explicit model*. An explicit model has the potential satisfaction of path formulas encoded as propositions. To check whether these so called encoded witnesses actually satisfy a path formula, is much easier than checking the basic formula (an ATL* formula starting with a path quantifier) directly. In fact, for synthesis, we do not need ATL* as a specification language. Using only explicit models, LTL is sufficient. Additionally, checking whether a model is an explicit model is theoretically possible in LTL plus a small HyperLTL part as well. Since explicit models are already trees, it is, in comparison to general concurrent game structures, relatively easy to construct computation trees from explicit models. This is helpful for the second part of this chapter, where we need computation trees as models of LTL specifications.

3.1.1 From Models to Basic Models

As a first step, we introduce propositions for each basic subformula in an ATL* specification. Let φ be an arbitrary ATL* formula, B_φ the set of basic subformulas in φ, and $B_{\varphi id} = \{b \mid \varphi_b \in B_\varphi\}$. We call a model $S = (A, Q, q_0, AP \uplus B_{\varphi id}, l, \vec{M}, \tau) \models \varphi$ basic if, for all basic subformulas $\varphi_b \in B_\varphi$, every state, which satisfies φ_b is labeled with b.

Lemma 3.1 ([Sch08])**.** *An ATL* formula is satisfiable iff it has a basic model.*

3.1.2 From (Basic) Models to Tree Models

For the following steps, we need the concurrent game structure to be a tree. Let $S = (A, Q, q_0, AP, l, \vec{M}, \tau)$ be a CGS. Let $\vec{M}$ to be a set of move vectors as defined for the concurrent game structure in ➔Subsection 2.5.1. S is called a concurrent game tree, if $Q = (\vec{M})^*$, $q_0 = \varepsilon$ and $\tau(q, d) = s \cdot d$. As shown in [Sch08], we are able to create a concurrent game tree for each concurrent game structure. We call this procedure unraveling: Let $S_u = (A, (\vec{M})^*, \varepsilon, AP, l \circ u, \vec{M}, \tau : s \cdot d)$ be the concurrent game tree, where the unraveling function $u : (\vec{M})^* \rightarrow Q$ is defined recursively: $u(\varepsilon) = q_0$ and $u(n) = q' \Rightarrow u(n \cdot d) = \tau(q', d)$.

➔Sec. 2.5, Page 11

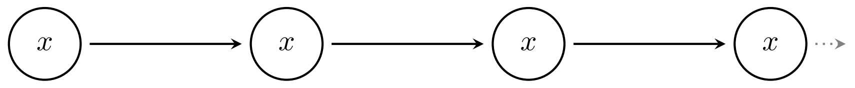
Lemma 3.2 ([Sch08])**.** *A concurrent game structure S is a (basic) model of a specification φ if and only if its unraveling S_u is a (basic) model of φ.*

Example 3.3 (Concurrent Game Trees)**.** As an example we use the running example from Figure 3.2a. We unravel the outgoing transitions of the state q_0. The concurrent game tree therefore starts with the node ε. The directions are based on the transition function from the concurrent game structure, hence the move vectors $\langle m_1, m_2 \rangle$ and $\langle m_1', m_2 \rangle$. According to this, the state q_1 unravels to the node εd_1. A visualization is given in Figure 3.2b. △

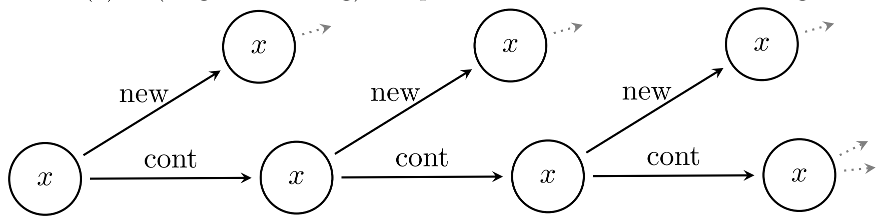
3.1.3 From Tree Models to Widened Tree Models

For a concurrent game tree $S = (A, Q, q_0, AP, l, \vec{M}, \tau)$, we construct a widened tree $S_w = (A, (\vec{M}')^*, q_0, AP, l \circ h, \vec{M}', \tau')$, where $\vec{M}' = \vec{M} \times \{new, cont\}$, $h : (\vec{M}')^* \rightarrow (\vec{M})^*$ is a hiding function which removes $\{new, cont\}$ from a trace, and $\tau'(q, d) = s \cdot d$ is the usual transition function. The widening is constructed to ensure that different witness strategies do not overlap, which is necessary to encode witness strategies in a finite amount of propositions per state.

Let us explain, how and why this widening allows us to encode an infinite amount of witnesses into the concurrent game tree. We encode witnesses state wise, therefore the goal is to have a finite amount of witnesses in each state.



(a) A (single-branching) computation tree before the widening



(b) The (single-branching) computation tree after the widening

Figure 3.1: A schematic example for the widening

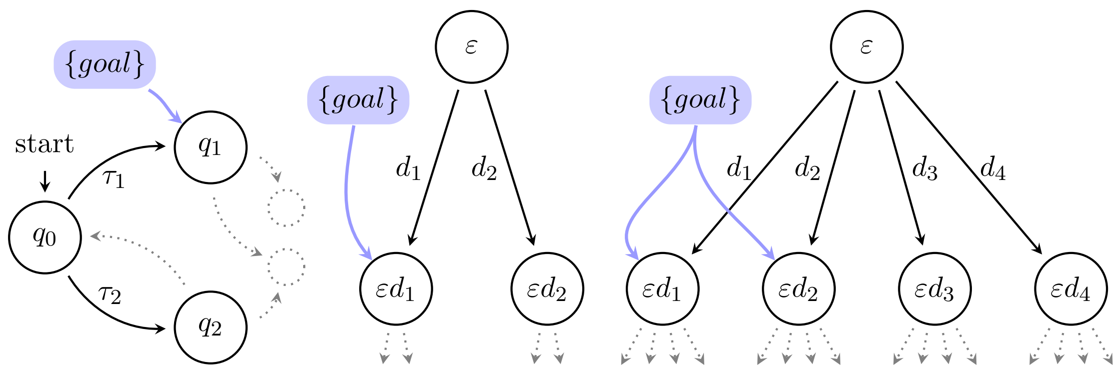
Assume that there are only two witnesses per agent, basic formula and state. We demonstrate a technique such that using this technique the widened tree using the bound of two is not more restricted than the original tree with no bound: We split the directions of each state into two halves, one with directions including the *cont* parameter, one including the *new* parameter . We call the first half *continue directions*, the second half *new directions*. Whenever we are at the start of an evaluation of a basic formula φ_b, hence a state which is labeled with b, we first go into a *new direction*, after that, hence whenever we are just traversing a state, we use the *continue direction*. Using this technique, in each state are at most two different witnesses per basic formula. One which is starting at that state and one which is traversing the state. Consider for Example Figure 3.1, which displays a (single-branching) computation tree in which $\langle\langle a \rangle\rangle \Box x$ holds. In Figure 3.1a, we would need to label a node in the infinite future, with infinitely many witnesses as they summarize for each node. In Figure 3.1b, however, due to the widening, by taking *new* after instancing a new witness and *cont* for continuation, only two witnesses exist in each node.

Lemma 3.4 ([Sch08])**.** *A concurrent game structure is a (basic) model of a specification φ if and only if its boolean widening is a (basic) model of φ.*

Example 3.5 (Widening of a Concurrent Game Tree)**.** In this example, we illustrate the widening of a concurrent game tree. Since we perform a boolean widening of the tree, each direction gets duplicated. We differentiate both, using $\{new, cont\}$. Each direction is a tuple of a move vector and *new* or *cont*. We denote that the first two directions originate from d_1, while the last both directions originate from d_2 of the not widened concurrent game tree. △

3.1.4 From Widened Tree Models to Explicit Models

In this section, we introduce several rules, which make a model an explicit model. If the following rules apply to an explicit model, we call it *well-formed*.



$\tau_1 \coloneqq \tau(q_0, \langle m_1, m_2\rangle)$

$\tau_2 \coloneqq \tau(q_0, \langle m_1', m_2\rangle)$

(a) A concurrent game structure.

$d_1 = \langle m_1, m_2\rangle$

$d_2 = \langle m_1', m_2\rangle$

(b) A concurrent game tree.

$d_1 = (\langle m_1, m_2\rangle, \mathit{new})$, $d_2 = (\cdots, \mathit{cont})$

$d_3 = (\langle m_1', m_2\rangle, \mathit{new})$, $d_4 = (\cdots, \mathit{cont})$

(c) The widening of a concurrent game tree

Figure 3.2: A concurrent game structure, its tree and widening.

Let for every basic subformula $\varphi_b \in B_\varphi$, $\mathit{agents}(b)$ the coalition of the basic subformula. With $\mathit{agents}(\neg b)$, we extract the complementary coalition: $\mathit{agents}(\neg b) = A \setminus \mathit{agents}(b)$, assuming A is the set of all agents.

Let us introduce an agent renumbering $e_b : A \to A$ for each basic subformula, with k the number of all agents. We then rearrange the agents according to their coalitions in the basic subformula. Let $j : 1 \leq j \leq k$, the number of agents in the coalition $\mid \mathit{agents}(b) \mid = j$. For each basic subformula φ_b, we introduce the function $e_b : \{1, \cdots k\} \to \{1, \cdots, k\}$, where we project the rearranged indexed to the original index. Therefore, the set of agents in the coalition are denoted as $\{a_{e_b(i)} \mid 1 \leq i \leq j\}$, all agents not in the coalition are in the set: $\{a_{e_b(i)} \mid j+1 \leq i \leq k\}$.

Additionally we introduce new propositions $E_\varphi = \{(b, \mathit{new}), (b, \mathit{cont}), (\neg b, \mathit{new}), (\neg b, \mathit{cont}) \mid \varphi_b \in B_\varphi\}$.

We call a concurrent game structure well-formed if for all basic subformulas, it satisfies the following requirements. For readability we omit b of the function e.

- $\rho_{b-new} \coloneqq \forall q \in Q.\, b \in l(q) \to$
  $\exists m_{e(1)} \in M^q_{a_{e(1)}} \cdots \exists m_{e(j)} \in M^q_{a_{e(j)}}.\, \forall m_{e(j+1)} \in M^q_{a_{e(j+1)}} \cdots \forall m_{e(k)} \in M^q_{a_{e(k)}}.$
  $(b, \mathit{new}) \in l(\tau(q, \langle m_1, \cdots, m_k\rangle))$
- $\rho_{new-cont} \coloneqq \forall q \in Q.\, (b, \mathit{new}) \in l(q) \to$
  $\exists m_{e(1)} \in M^q_{a_{e(1)}} \cdots \exists m_{e(j)} \in M^q_{a_{e(j)}}.\, \forall m_{e(j+1)} \in M^q_{a_{e(j+1)}} \cdots \forall m_{e(k)} \in M^q_{a_{e(k)}}.$
  $(b, \mathit{cont}) \in l(\tau(q, \langle m_1, \cdots, m_k\rangle))$
- $\rho_{cont-cont} \coloneqq \forall q \in Q.\, (b, \mathit{cont}) \in l(q) \to$
  $\exists m_{e(1)} \in M^q_{a_{e(1)}} \cdots \exists m_{e(j)} \in M^q_{a_{e(j)}}.\, \forall m_{e(j+1)} \in M^q_{a_{e(j+1)}} \cdots \forall m_{e(k)} \in M^q_{a_{e(k)}}.$
  $(b, \mathit{cont}) \in l(\tau(q, \langle m_1, \cdots, m_k\rangle))$

- $\rho_{not(b-new)} \coloneqq \forall q \in Q.\, b \notin l(q) \rightarrow$
  $\forall m_{e(1)} \in M^q_{a_{e(1)}} \cdots \forall m_{e(j)} \in M^q_{a_{e(j)}}. \exists m_{e(j+1)} \in M^q_{a_{e(j+1)}} \cdots \exists m_{e(k)} \in M^q_{a_{e(k)}}.$
  $(\neg b, new) \in l(\tau(q, \langle m_1, \cdots, m_k \rangle))$
- $\rho_{not(new-cont)} \coloneqq \forall q \in Q.\, (\neg b, new) \in l(q) \rightarrow$
  $\forall m_{e(1)} \in M^q_{a_{e(1)}} \cdots \forall m_{e(j)} \in M^q_{a_{e(j)}}. \exists m_{e(j+1)} \in M^q_{a_{e(j+1)}} \cdots \exists m_{e(k)} \in M^q_{a_{e(k)}}.$
  $(\neg b, cont) \in l(\tau(q, \langle m_1, \cdots, m_k \rangle))$
- $\rho_{not(cont-cont)} \coloneqq \forall q \in Q.\, (\neg b, cont) \in l(q) \rightarrow$
  $\forall m_{e(1)} \in M^q_{a_{e(1)}} \cdots \forall m_{e(j)} \in M^q_{a_{e(j)}}. \exists m_{e(j+1)} \in M^q_{a_{e(j+1)}} \cdots \exists m_{e(k)} \in M^q_{a_{e(k)}}.$
  $(\neg b, cont) \in l(\tau(q, \langle m_1, \cdots, m_k \rangle))$

Using these requirements, we implicitly enforce an encoding of witness strategies and witness counter strategies. For explanation, let us analyze the first requirement ρ_{b-new}: In each state which is marked with the proposition indicating a basic formula b, we require that some of the successor states are labeled with (b, new). Which successor states are intended, is specified in the second line through the move vector. Each agent contributes one move to the move vector. The move vector then determines the transition. For the agents in the coalition ($a_{e(1)}$ to $a_{e(j)}$) there has to be some move per agent, whereas for all other agents ($a_{e(j+1)}$ to $a_{e(k)}$) we take each of their moves. By quantifying the moves of the agent, the move vectors and therefore the transitions are quantified indirectly. Concluded, a successor state is labeled if it can be reached by some move per agent from the coalition, but each move from all other agents. The other requirements are similar, for counter strategies we require the opposite move quantification. Assuming that the proposition b indicates the state where the path formula will be evaluated, these requirements enforce the implicit encoding of plays into the system. We introduced the requirement that each witness chooses the *new* proposition in its first step, to ensure that only two propositions are sufficient to encode all possible witnesses.

The following formula describes how to extract the plays or witnesses from the system. For a basic formula $\varphi_b \in B_\varphi$, we call a set of traces $witness(q, b) = \{q q_1 q_2 q_3 \cdots \mid b \in l(q), (b, new) \in l(q_1), \forall i \geq 2.\, (b, cont) \in l(q_i)\}$ a witness of the satisfaction of φ_b. Analogously, $witness(q, \neg b) = \{q q_1 q_2 q_3 \cdots \mid b \notin l(q), (\neg b, new) \in l(q_1), \forall i \geq 2.\, (\neg b, cont) \in l(q_i)\}$ a witness of the not-satisfaction of φ_b.

We call a model an explicit model $S = (A, (\vec{M}')^*, q_0, AP \uplus B_{\varphi id} \uplus E_\varphi, l \circ h, \vec{M}', \tau')$ of an ATL* formula φ, if the explicit witnesses of all basic subformulas are contained in the set of paths that satisfy φ_b and $\neg\varphi_b$ respectively.

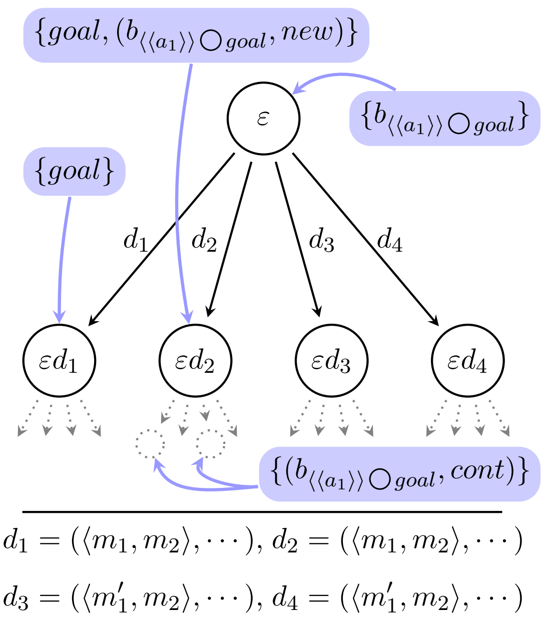
Lemma 3.6 ([Sch08])**.** *Given a concurrent game tree S that is a basic model of an ATL* formula φ and a set of witness strategies for S, we can construct an explicit model of φ.*

Example 3.7 (Explicit Models)**.** In this example, we sketch, how a concurrent game tree becomes well-formed. Let us only consider the root node. First, we label the root node with the proposition $b_{\langle\langle a_1 \rangle\rangle \bigcirc goal}$, which indicats the evaluation start of the basic formula $\langle\langle a_1 \rangle\rangle \bigcirc goal$. A sketch of this figure can be found in Figure 3.3a. To be well formed, in this example, the first line of the well-formedness requirements is important. Concerning the basic formula $\langle\langle a_1 \rangle\rangle \bigcirc goal$, and state q_0 of our example, this reduces to

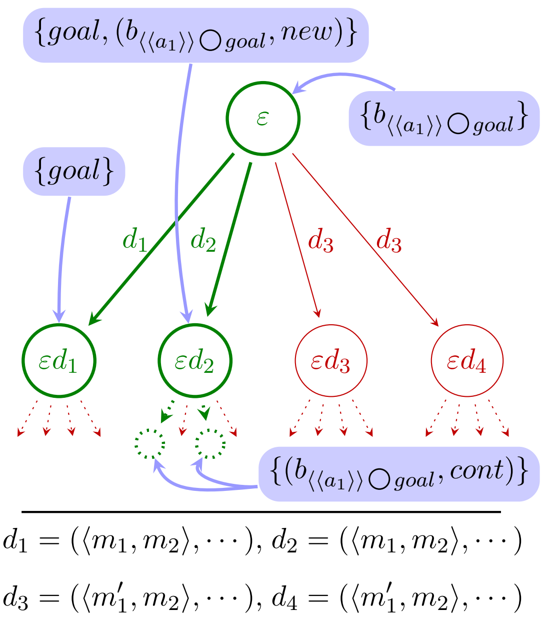
$$\exists m \in \{m_1, m_1'\}. \forall m' \in \{m_2\}.\, (b, new) \in l(\tau(q_0, \langle m, m' \rangle)).$$

We therefore choose to label the node εd_2 with the proposition (b, new), such that the formula above is satisfied and the system is well-formed. Note that, because of the hiding function, we cannot distinguish between εd_1 and εd_2. Labeling εd_1 or εd_2 or both, makes no actual difference. The same holds for εd_3 and εd_4. Additionally to be well-formed, the successor of εd_1 and εd_2 have to be partly labeled with $(b, cont)$ and so on. For this example, we only implied these labels and future of the tree.

In Figure 3.3b, you can see the well-formed concurrent game structure with marked witnesses. The witness is extracted from the structure using this formula: $witness(q, b) = \{qq_1q_2q_3 \cdots \mid b \in l(q), (b, new) \in l(q_1), \forall i \geq 2. (b, cont) \in l(q_i)\}$. For the thick green paths this formula holds, hence these are witnesses for the formula $\langle\langle a_1 \rangle\rangle \bigcirc goal$. Consequently the shown concurrent game structure is an explicit model of $\langle\langle a_1 \rangle\rangle \bigcirc goal$. $\triangle$


$\{goal, (b_{\langle\langle a_1\rangle\rangle \bigcirc goal}, new)\}$
ε
$\{b_{\langle\langle a_1\rangle\rangle \bigcirc goal}\}$
$\{goal\}$
d_1
d_2
d_3
d_4
εd_1
εd_2
εd_3
εd_4
$\{(b_{\langle\langle a_1\rangle\rangle \bigcirc goal}, cont)\}$
$d_1 = (\langle m_1, m_2\rangle, \cdots)$, $d_2 = (\langle m_1, m_2\rangle, \cdots)$
$d_3 = (\langle m_1', m_2\rangle, \cdots)$, $d_4 = (\langle m_1', m_2\rangle, \cdots)$

(a) A well-formed concurrent game tree.


$\{goal, (b_{\langle\langle a_1\rangle\rangle \bigcirc goal}, new)\}$
ε
$\{b_{\langle\langle a_1\rangle\rangle \bigcirc goal}\}$
$\{goal\}$
d_1
d_2
d_3
d_3
εd_1
εd_2
εd_3
εd_4
$\{(b_{\langle\langle a_1\rangle\rangle \bigcirc goal}, cont)\}$
$d_1 = (\langle m_1, m_2\rangle, \cdots)$, $d_2 = (\langle m_1, m_2\rangle, \cdots)$
$d_3 = (\langle m_1', m_2\rangle, \cdots)$, $d_4 = (\langle m_1', m_2\rangle, \cdots)$

(b) A concurrent game tree with marked witnesses.

Figure 3.3: An explicit model of the formula $\langle\langle a_1 \rangle\rangle \bigcirc goal$.

3.1.5 Realizability of Explicit models

We use the results of [Sch08] to argue that each ATL* specification is realizable using only explicit models.

Theorem 3.8 ([Sch08])**.** *A specification has an explicit model if and only if it has a model.*

As a direct result of Theorem 3.8 we conclude that every specification, which is realizable, is also realizable if we limit ourself to only explicit models. Also, each specification which is realizable by an explicit model, can also be realized by a general concurrent game structure.

3.2 ATL* Reduction

We now use the findings of [Sch08], explained in the previous Section 3.1, to reduce ATL* synthesis to (Hyper-)LTL synthesis. As it turns out, after some additions to explicit models, an LTL formula plus a small HyperLTL part is sufficient to test whether such a model satisfies an ATL* specification. Additionally, we are able to test whether a model is an explicit model using only LTL as well. Consequently, we show that the constructed (Hyper-)LTL formula is realizable if and only if the ATL* specification is realizable.

As already mentioned, the goal is to construct a (Hyper-)LTL formula which is sufficient to synthesize an ATL* specification. (Hyper-)LTL synthesis however constructs a trace model as result of the synthesis algorithm, for ATL* synthesis we require the model to be a concurrent game structure, more precisely w.l.o.g. an explicit model. As the trace semantics of LTL and HyperLTL can be easily transformed into computation trees as shown in Subsection 2.3.2, we show how an explicit model can be transformed into a computation tree.

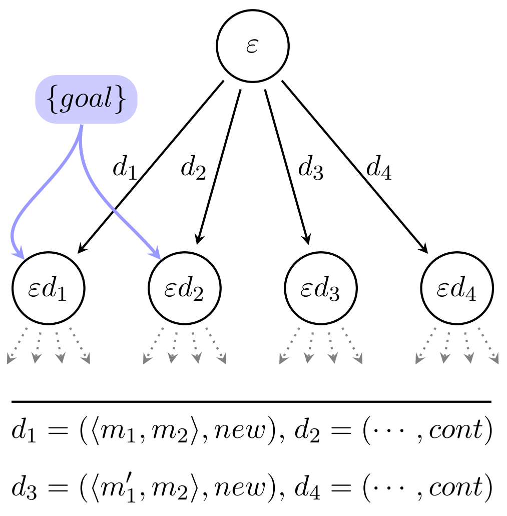
In contrary to concurrent game trees, a computation tree is spanned by input variables, not move vectors. As a transformation, we use a mapping from move vectors to sets of input variables. Further, we encode the membership of moves in a move vector, to be able to specify requirements concerning specific moves. At this point we also need to make sure that only one move per agent is part of a move vector. We call this *contribution requirement* because it requires that each agent contributes to the move vector, hence the decision which state is next. Finally, we construct LTL formulas for the well-formedness criteria and the satisfaction of basic formulas. All of this will be assembled to one LTL specification with a small HyperLTL part, which then solves the synthesis problem for ATL*.

3.2.1 Encoding Inputs

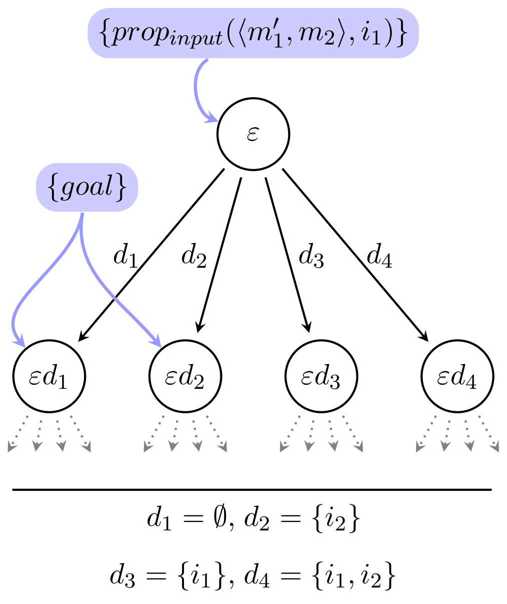
We introduce a proposition to encode a mapping from trees, spanned over move vectors, to trees spanned over input variables. Having the mapping implemented we are able to span the concurrent game trees over input variables instead of move vectors. As described in ➔Subsection 3.1.3, the concurrent game tree is widened. This is solved by introducing an additional input variable i_{new}. However, this input variable will only be used to span the tree. We will not use the new input variable as part of the mapping, such that it makes it impossible to distinguish directions with or without the input variable i_{new}. Consequently, whenever we will talk about a set of inputs I, this does not include the new input variable: $i_{new} \notin I$. However, this is not sufficient to implement the hiding function, as we will show in Subsection 3.2.3.

➔Sec. 3.1, Page 20

We add the new proposition to every node q of the computation tree.


ε
$\{goal\}$
d_1
d_2
d_3
d_4
εd_1
εd_2
εd_3
εd_4
$d_1 = (\langle m_1, m_2\rangle, new)$, $d_2 = (\cdots, cont)$
$d_3 = (\langle m_1', m_2\rangle, new)$, $d_4 = (\cdots, cont)$

(a) A widened concurrent game tree.


$\{prop_{input}(\langle m_1', m_2\rangle, i_1)\}$
ε
$\{goal\}$
d_1
d_2
d_3
d_4
εd_1
εd_2
εd_3
εd_4
$d_1 = \emptyset$, $d_2 = \{i_2\}$
$d_3 = \{i_1\}$, $d_4 = \{i_1, i_2\}$

(b) A widened concurrent game tree, spanned by inputs.

Figure 3.4: A concurrent game tree spanned by moves vectors versus inputs.

- $prop_{input}(\vec{m}, i)$, for every $\vec{m} \in \vec{M}^q$ and $i \in I$. Then

$$prop_{input}(\vec{m}, i) \text{ if and only if } i \text{ belongs to } \vec{m}$$

Example 3.9 (Introducing Inputs)**.** In this example, we illustrate how concurrent game trees are transformed to trees, which are spanned by inputs, not move vectors. We first introduce input variable i_1, which for the state q_0 differentiates the move vectors $\langle m_1, m_2\rangle$ and $\langle m_1', m_2\rangle$. Going into a direction marked with $\langle m_1, m_2\rangle$ is equal to setting $\neg i_1$, going towards $\langle m_1', m_2\rangle$ is equal to setting i_1. Secondly, we introduce a second input variable i_2. This input variable is not considered in the introduced propositions as its only purpose is to widen the tree. As required by the hiding function it is not possible to distinguish between between the directions $\emptyset$ and $\{i_2\}$, respectively $\{i_1\}$ and $\{i_1, i_2\}$. A visualization is given in Figure 3.4. △

Following Move Vectors

In the case of synthesis, we can assume that input values are set by the synthesizer. Further we can assume that the synthesizer sets each combination of input values at some point. Following the idea presented in [BSK17], we react to the input values set by the synthesizer to recognize the direction / move vector which is currently selected. While in LTL it is regularly only possible to select all traces, hence paths, this trick makes it possible to select certain paths from a computation tree using LTL and in consequence require different conditions on these paths. We call this following a move vector.

We are able to decide whether we follow a certain move vector $\vec{m}$ through this formula:

$$\varphi_{follow}(\vec{m}) := \bigwedge_{i \in I} prop_{input}(\vec{m}, i) = i$$

Example 3.10 (Following Move Vectors)**.** As the input variables are set by the synthesizer, we assume for this example $i_1 = true, i_2 = true$. For the move vector $\langle m_1, m_2 \rangle$ we get the following:

$$\bigwedge_{i \in I} prop_{input}(\langle m_1, m_2 \rangle, i) = i \;\equiv\; prop_{input}(\langle m_1, m_2 \rangle, i_1) = i_1$$

$$\equiv false = true \;\equiv\; false$$

Whereas for the move vector $\langle m_1', m_2 \rangle$ we get:

$$\bigwedge_{i \in I} prop_{input}(\langle m_1, m_2 \rangle, i) = i \;\equiv\; prop_{input}(\langle m_1, m_2 \rangle, i_1) = i_1$$

$$\equiv true = true \;\equiv\; true$$

Therefore, we know that we are following the move vector $\langle m_1, m_2 \rangle$.

Note that we did not consider the input variable i_2 at all. The reason for this is that we defined i_2 to be the additionally added input variable which only purpose is to widen the concurrent game tree. △

3.2.2 Encoding Moves

We encode the set of available moves per agent and state, as well as the membership of moves in a move vector through similar propositions as in Subsection 3.2.1. For every state q we introduce:

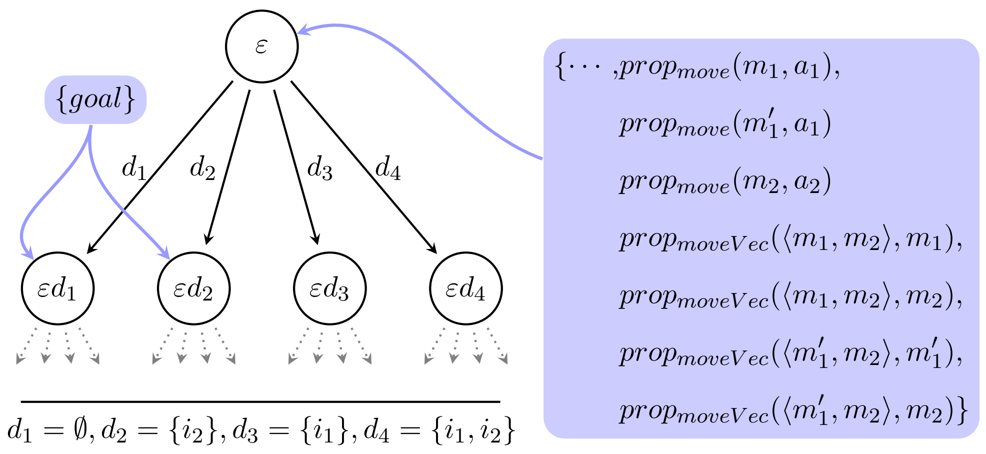
- $prop_{move}(m, a)$, for every possible move m and $a \in A$. Then

  $$prop_{move}(m, a) \text{ if and only if } m \in M_a^q$$

- $prop_{moveVec}(\vec{m}, m)$, for every $\vec{m} \in \vec{M}$, $a \in A = \{a_1, \cdots, a_k\}$ and $m \in M_a^q$. Then

  $$prop_{moveVec}(\langle m_1, \cdots, m_k \rangle, m) \text{ if and only if } \forall i \in \{1, \cdots, k\}.\, m_i \in M_{a_i}^n$$

Example 3.11 (Introducing Propositions for Moves and Move Vectors)**.** We give an example of the labeling regarding moves and move vectors of the node q_0 from our running example in Figure 3.5. With $prop_{move}(m, a)$, we encode the belonging of moves to agents. In our example m_1 and m_1' belong to a_1 and m_2 belongs to a_2. Further, we encode which moves belong to which move vectors. △



ε
$\{goal\}$
d_1
d_2
d_3
d_4
εd_1
εd_2
εd_3
εd_4
$d_1 = \emptyset, d_2 = \{i_2\}, d_3 = \{i_1\}, d_4 = \{i_1, i_2\}$
$\{\cdots, prop_{move}(m_1, a_1),$
$prop_{move}(m'_1, a_1)$
$prop_{move}(m_2, a_2)$
$prop_{moveVec}(\langle m_1, m_2\rangle, m_1),$
$prop_{moveVec}(\langle m_1, m_2\rangle, m_2),$
$prop_{moveVec}(\langle m'_1, m_2\rangle, m'_1),$
$prop_{moveVec}(\langle m'_1, m_2\rangle, m_2)\}$

Figure 3.5: A concurrent game tree with proposition for moves and move vectors

Contribution Requirement

As mentioned, we require that each agent contributes exactly one move to a move vector. We test this condition with the following formula:

$$\forall q \in Q. \forall \vec{m} \in \vec{M}. \forall a \in A. \exists m_i \in M^q_{a_i}. \forall m'_i \in M^q_{a_i}. (m'_i \in \vec{m} \leftrightarrow m_i = m'_i)$$

We are able to translate this directly into LTL:

$$\varphi_{contr} := \Box \bigwedge_{\vec{m}\in\vec{M}} \bigwedge_{a\in A} \bigvee_{m\in M_a} \bigwedge_{m'\in M_a} prop_{move}(m,a) \land prop_{move}(m',a) \rightarrow$$
$$(prop_{moveVec}(\vec{m}, m') \leftrightarrow m = m')$$

We define $m = m'$ in LTL as the bitwise comparison of m and m'.

Example 3.12 (Contribution Requirement)**.** For a better understanding of the contribution requirement, assume that the propositions $prop_{moveVec}(\langle m_1, m'_1\rangle, m_1)$ and $prop_{moveVec}(\langle m_1, m'_1\rangle, m'_1)$ hold. Since we defined $prop_{move}(m_1, a_1)$ and $prop_{move}(m_1, a_1)$ as seen in Figure 3.5, according to φ_{contr}, m_1 has to be equal to m'_1, which is not the case since they produce different behavior. Consequently, it is not possible that both propositions hold. △

3.2.3 Behavior Equality

Before we introduce witnesses, we implement the second part of the hiding function from ➔Subsection 3.1.3. As mentioned before, the first part of implementing the hiding function is done by hiding the new input i_{new} from the propositions. Nevertheless The

➔Sec. 3.1, Page 20

LTL synthesizer still uses the new input variable for the branching, hence given two directions, which are equivalent under the hiding function are leading to two different nodes. Besides the fact that, based on the propositions, we do not know which direction was used to get to these nodes, we still are in different nodes, which may lead to a different evaluation of the formula. Because of this, we introduce behavior equality. Two traces are behavior equal, if all propositions in each state, besides witnesses, are equal. We introduce a small HyperLTL formula, which enforces behavior equality between widened traces, hence traces, which have, up to the new input variable, the same input sequences. That way, we do not really hide the new input variable, but enforce the same behavior on traces which originated from a single trace.

Let I be the set of input variables without the newly introduced proposition. Let the set $AP \setminus witness$, be the set of all atomic proposition we introduced up to this section. $AP \setminus witness = \{prop_{input}(\vec{m}, i) \mid \vec{m} \in \vec{M}, i \in I\} \cup \{prop_{move}(m, a) \mid a \in A, m \in M_a\} \cup \{prop_{moveVec}(\vec{m}, m) \mid \vec{m} \in \vec{M}, a \in A, m \in M_a\}$.

$$\varphi_{behavior-equal} = \forall \pi, \pi'. \square((\bigwedge_{i \in I} i_\pi \leftrightarrow i_{\pi'}) \rightarrow (\bigwedge_{ap \in AP \setminus witness} ap_\pi \leftrightarrow ap_{\pi'}))$$

3.2.4 Well-Formedness

In this section, we implement the well-formedness formulas from ➔Subsection 3.1.4 in LTL. These formulas indirectly mark the witnesses of satisfaction for each basic formula. ➔Sec. 3.1, Page 21

We consider three parts of the well-formedness formulas independently. Firstly, some of the moves be universally quantified, secondly some moves are quantified existentially. The third part is implementing the transition function.

We first look at the part of universally and existentially quantified moves. Let $\{a_1, \cdots, a_j\}$ be a subset of the agents $\{a_1, \cdots, a_k\}$. We construct the following formula in LTL:

$$\varphi_{\forall moves}(a_1, \cdots, a_j) := \bigwedge_{\substack{m_1 \in M \\ \vdots \\ m_j \in M}} prop_{move}(m_1, a_1) \wedge \cdots \wedge prop_{move}(m_j, a_j)$$

$$\varphi_{\exists moves}(a_1, \cdots, a_j) := \bigvee_{\substack{m_1 \in M \\ \vdots \\ m_j \in M}} prop_{move}(m_1, a_1) \wedge \cdots \wedge prop_{move}(m_j, a_j)$$

The next formula is constructed to implicitly encode transitions of concurrent game structures into computation trees. Let x be some proposition and $m_1, \cdots, m_k$ be moves by the agents $1, \cdots, k$ respectively. First, we detect which move vector leads to the next state, by comparing move vectors and input variables set by the synthesizer. We use the formula φ_{follow} combined with the disjunction to detect the correct move vector.

Further, we test if the detected move vector matches the chosen moves of the agents. Whenever this is the case, the next state has to have a specific label x. As defined in ➔Subsection 3.1.3, a hiding function is applied to the labeling function, hence as next states we do not consider all next states but one of two states: One state, we reached using the fresh introduced input variable i_{new}, one state without. As a parameter for this distinction, we take a placeholder variable *new*, on which boolean value we decide, whether we consider a direction with or without i_{new}.

➔Sec. 3.1, Page 20

$$\varphi_{trans}(x, m_1, \cdots, m_k, new) :=$$
$$\bigwedge_{\vec{m} \in \vec{M}} (\varphi_{follow}(\vec{m}) \land prop_{moveVec}(\vec{m}, m_1) \land \cdots \land prop_{moveVec}(\vec{m}, m_k)) \land$$
$$new = i_{new} \rightarrow \bigcirc x$$

Based on these subformulas we are able to construct all the well-formedness constraints from ➔Subsection 3.1.4 . As before, we use the renumbering e_b for a basic subformula φ_b of φ. Please note that the moves $m_1 \cdots m_k$ are quantified in $\varphi_{\exists moves}$ respectively $\varphi_{\forall moves}$. As explained in ➔Subsection 3.1.3, we use the presented technique to decide whether the next state we will label, is reached through a *continue direction* or *new direction.*

➔Sec. 3.1, Page 21

➔Sec. 3.1, Page 20

- $\square(b \rightarrow \varphi_{\exists moves}(a_{e_b(1)}, \cdots, a_{e_b(j)}) \land \varphi_{\forall moves}(a_{e_b(j+1)}, \cdots, a_{e_b(k)}) \rightarrow$ $\varphi_{trans}((b, new), m_1, \cdots, m_k, true))$
- $\square((b, new) \rightarrow \varphi_{\exists moves}(a_{e_b(1)}, \cdots, a_{e_b(j)}) \land \varphi_{\forall moves}(a_{e_b(j+1)}, \cdots, a_{e_b(k)}) \rightarrow$ $\varphi_{trans}((b, cont), m_1, \cdots, m_k, false))$
- $\square((b, cont) \rightarrow \varphi_{\exists moves}(a_{e_b(1)}, \cdots, a_{e_b(j)}) \land \varphi_{\forall moves}(a_{e_b(j+1)}, \cdots, a_{e_b(k)}) \rightarrow$ $\varphi_{trans}((b, cont), m_1, \cdots, m_k, false))$
- $\square(\neg b \rightarrow \varphi_{\forall moves}(a_{e_b(1)}, \cdots, a_{e_b(j)}) \rightarrow \varphi_{\exists moves}(a_{e_b(j+1)}, \cdots, a_{e_b(k)}) \land$ $\varphi_{trans}((\neg b, new), m_1, \cdots, m_k, true))$
- $\square((\neg b, new) \rightarrow \varphi_{\forall moves}(a_{e_b(1)}, \cdots, a_{e_b(j)}) \rightarrow \varphi_{\exists moves}(a_{e_b(j+1)}, \cdots, a_{e_b(k)}) \land$ $\varphi_{trans}((\neg b, cont), m_1, \cdots, m_k, false))$
- $\square((\neg b, cont) \rightarrow \varphi_{\forall moves}(a_{e_b(1)}, \cdots, a_{e_b(j)}) \rightarrow \varphi_{\exists moves}(a_{e_b(j+1)}, \cdots, a_{e_b(k)}) \land$ $\varphi_{trans}((\neg b, cont), m_1, \cdots, m_k, false))$

We denote the conjunction of all these formulas with $\varphi_{well-formed}(b)$.

Example 3.13 (Well-formedness Using Propositions)**.** In Figure 3.6, we repeated the illustration from our running example with some of the propositions we added since now. We argue, why this tree is well-formed according to the LTL definition of well-formedness. As before in ➔Example 3.7, we consider only the first line of the well-formedness requirement. For the basic formula $\langle\langle a_1 \rangle\rangle \bigcirc goal$ and the state q_0 in our example, where $b_{\langle\langle a_1 \rangle\rangle \bigcirc goal}$ directly holds, this transforms to

➔Sec. 3.1, Page 23

$$
\begin{aligned}
&(prop_{move}(m_1, a_1) \wedge prop_{move}(m_2, a_2)) \rightarrow (\\
&\quad (\varphi_{follow}(\langle m_1, m_2\rangle) \wedge prop_{moveVec}(\langle m_1, m_2\rangle, m_1) \wedge prop_{moveVec}(\langle m_1, m_2\rangle, m_2)) \wedge \\
&\qquad i_{new} \rightarrow \bigcirc b_{\langle\langle a_1\rangle\rangle \bigcirc goal}) \wedge \\
&\quad (\varphi_{follow}(\langle m_1', m_2\rangle) \wedge prop_{moveVec}(\langle m_1', m_2\rangle, m_1) \wedge prop_{moveVec}(\langle m_1', m_2\rangle, m_2)) \wedge \\
&\qquad i_{new} \rightarrow \bigcirc b_{\langle\langle a_1\rangle\rangle \bigcirc goal}) \\
&) \vee (prop_{move}(m_1', a_1) \wedge prop_{move}(m_2, a_2)) \rightarrow (\\
&\quad (\varphi_{follow}(\langle m_1, m_2\rangle) \wedge prop_{moveVec}(\langle m_1, m_2\rangle, m_1') \wedge prop_{moveVec}(\langle m_1, m_2\rangle, m_2)) \wedge \\
&\qquad i_{new} \rightarrow \bigcirc b_{\langle\langle a_1\rangle\rangle \bigcirc goal}) \wedge \\
&\quad (\varphi_{follow}(\langle m_1', m_2\rangle) \wedge prop_{moveVec}(\langle m_1', m_2\rangle, m_1') \wedge prop_{moveVec}(\langle m_1', m_2\rangle, m_2)) \wedge \\
&\qquad i_{new} \rightarrow \bigcirc b_{\langle\langle a_1\rangle\rangle \bigcirc goal}) \\
&)
\end{aligned}
$$

This formula consists of two parts. First and second half originate from quantification of moves, hence the disjunction of a moves from agent a_1 and the conjunction of moves from agent a_2 (which is only a single one, hence no conjunction). Inside these halves, the formula consists of two parts respectively which originate from the conjunction of possible move vectors. Inside that we have the instances of the formula φ_{trans}.

Testing this formula on Figure 3.6, we find out that the concurrent game tree is well-formed.

$\triangle$

3.2.5 Path Satisfaction

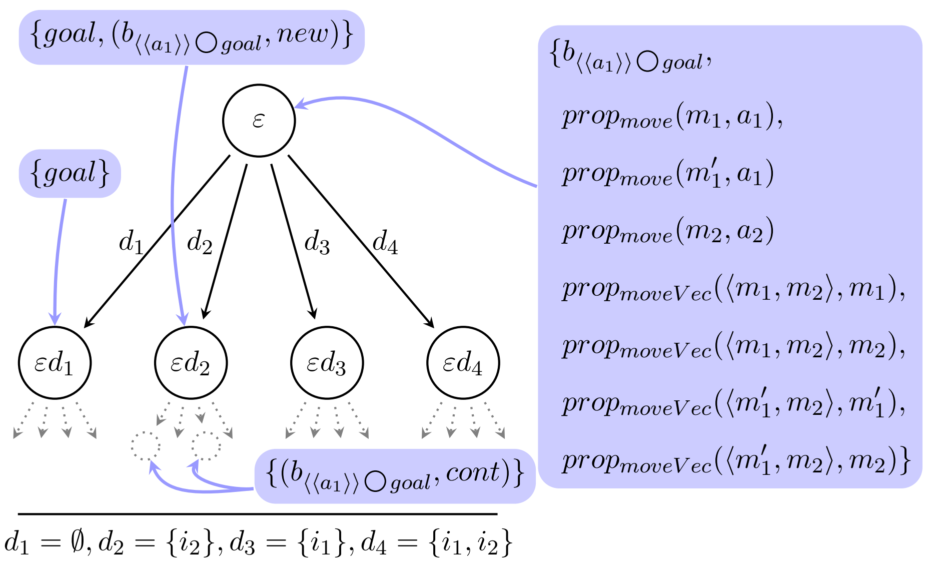
In this section, we implement the witness recognition and path satisfaction.

We first create a formula, which is recognizing witnesses in the explicit model. Let φ_b be a subformula of the ATL* formula φ, where b is the indicator for this subformula.

$$\varphi_{witness}(b) := b \wedge \bigcirc((b, new) \bigcirc \Box(b, cont))$$

$$\varphi_{cwitness}(b) := \neg b \wedge \bigcirc((\neg b, new) \bigcirc \Box(\neg b, cont))$$

Given an ATL* formula φ, we can replace all basic subformulas φ_b of φ with the proposition b. We denote the modified formula φ'. Consequently, let φ_b the subformula, where each basic subformula in φ_b is replaced in the same way. The satisfaction of path formulas is defined as follow:



$\{goal, (b_{\langle\langle a_1\rangle\rangle \bigcirc goal}, new)\}$
$\{goal\}$
ε
d_1
d_2
d_3
d_4
εd_1
εd_2
εd_3
εd_4
$\{(b_{\langle\langle a_1\rangle\rangle \bigcirc goal}, cont)\}$
$\{b_{\langle\langle a_1\rangle\rangle \bigcirc goal}, prop_{move}(m_1, a_1), prop_{move}(m_1', a_1), prop_{move}(m_2, a_2), prop_{moveVec}(\langle m_1, m_2\rangle, m_1), prop_{moveVec}(\langle m_1, m_2\rangle, m_2), prop_{moveVec}(\langle m_1', m_2\rangle, m_1'), prop_{moveVec}(\langle m_1', m_2\rangle, m_2)\}$
$d_1 = \emptyset, d_2 = \{i_2\}, d_3 = \{i_1\}, d_4 = \{i_1, i_2\}$

Figure 3.6: A well-formed concurrent game tree with move and move vector propositions.

$$\varphi_{satisfaction}(\varphi_b) \coloneqq (\varphi_{witness}(b) \rightarrow \varphi_b') \wedge (\varphi_{cwitness}(b) \rightarrow \neg\varphi_b')$$

Example 3.14 (Explicit Models Using Propositions)**.** In this example, we repeat the well-formed concurrent game tree with move and move vector propositions (Figure 3.6), but analyze witnesses and path satisfaction. A path that is a witness has to start with b, in the case of our example $b_{\langle\langle a_1\rangle\rangle \bigcirc goal}$. Then for the next state $(b_{\langle\langle a_1\rangle\rangle \bigcirc goal}, new)$ and for all further states $(b_{\langle\langle a_1\rangle\rangle \bigcirc goal}, cont)$. We illustrated witness paths in Figure 3.7. For our example, we only analyzed the first and second state. For further states, we assumed that the witness path continues correctly. Shown in Figure 3.14, we marked the witnesses with thick green.

Further, accordingly to the formula $\varphi_{satisfaction}$ we can test if on all of the witnesses, the body of the basic formula holds. For our example, the formula was $\langle\langle a_1\rangle\rangle \bigcirc goal$, hence the body is $\bigcirc goal$. As you can see, this holds on all witnesses.

Notice that the witness and consequently the satisfaction formula is not enforced on the path $\varepsilon \cdot d_1 \cdots$, despite that both originated from the same move vector. Having different satisfaction criteria on these branches, may lead to systems realizing unrealizable specifications, which is of course a contradiction. That is the reason, we introduced the HyperLTL formula in Subsection 3.2.3, enforcing behavior equality on such branches. From a synthesis view, the goal is to find the correct system, hence finding propositions, such that the satisfaction formula is satisfied. The HyperLTL formula enforces, that each setting of propositions (move vectors, moves and regular propositions), resulting from satisfying the satisfaction formula in one of the branches, also have to be set on the behavior equivalent branch. For our example this means, despite that witnesses are only marked in branch $\varepsilon \cdot d_2 \cdots$, node εd_1 also has to have the proposition *goal*. △

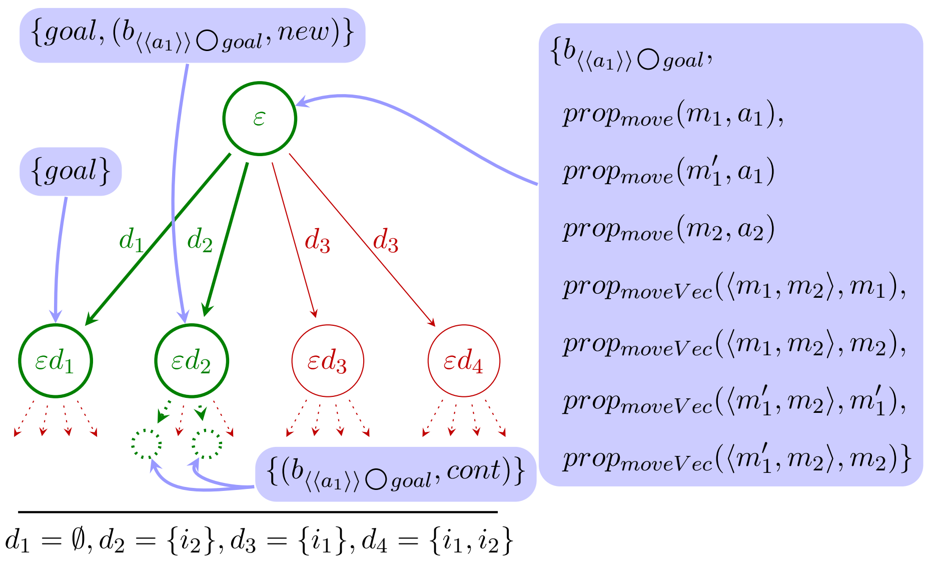

$\{goal, (b_{\langle\langle a_1\rangle\rangle\bigcirc goal}, new)\}$
$\{goal\}$
ε
d_1
d_2
d_3
d_3
εd_1
εd_2
εd_3
εd_4
$\{(b_{\langle\langle a_1\rangle\rangle\bigcirc goal}, cont)\}$
$\{b_{\langle\langle a_1\rangle\rangle\bigcirc goal},$
$prop_{move}(m_1, a_1),$
$prop_{move}(m_1', a_1)$
$prop_{move}(m_2, a_2)$
$prop_{moveVec}(\langle m_1, m_2\rangle, m_1),$
$prop_{moveVec}(\langle m_1, m_2\rangle, m_2),$
$prop_{moveVec}(\langle m_1', m_2\rangle, m_1'),$
$prop_{moveVec}(\langle m_1', m_2\rangle, m_2)\}$
$d_1 = \emptyset, d_2 = \{i_2\}, d_3 = \{i_1\}, d_4 = \{i_1, i_2\}$

Figure 3.7: A well-formed concurrent game tree with move and move vector propositions and recognized witnesses.

3.2.6 Complete LTL Formula

Let φ be some ATL* formula, let B_φ be the set of all basic subformulas in the formula φ. We denote a basic subformula as $\varphi_b \in B_\varphi$, where b is a unique identifier for this subformula. As mentioned before, given an ATL* formula φ we can replace all basic subformulas φ_b of φ with the proposition b. We denote the modified formula φ'.

$$\varphi_{behavior-equal} \wedge \varphi_{contr} \wedge \varphi' \wedge \bigwedge_{\varphi_b \in B_\varphi} \varphi_{well-formed}(b) \wedge \varphi_{satisfaction}(\varphi_b)$$

3.2.7 Complexity of the Constructed Formula

We analyze the complexity of the new formula in comparison to the ATL* formula.

Let us first look at the HyperLTL formula $\varphi_{behavior-equal}$. For this kind of formula the bounded synthesis of HyperLTL has a quadratic blowup [Fin+18] in comparison to LTL synthesis. Let us analyze the length of this formula. The length is dependent on the number of propositions we compare. For $prop_{input}(\vec{m}, i)$, we get for all combinations of move vectors $\vec{m}$ and inputs i, $\mathcal{O}(|I| \times 2^{|I|})$ propositions to compare. For $prop_{move}(m, a)$, we get for all combinations of moves m and agents a, $\mathcal{O}(2^{|I|} \times |A|)$ propositions. For $prop_{moveVec}(\vec{m}, m)$, we get for the combinations of move vectors $\vec{m}$ and moves m, $\mathcal{O}(2^{|I|} \times |A| \times 2^{|I|})$. For the whole formula we are in $\mathcal{O}(2^{2\times|I|} \times |A|)$ propositions, hence the length of the formula.

Secondly we analyze the subformula φ_{contr}. Based on the con- and disjunction in this formula, the length of φ_{contr} has the following complexity: $\mathcal{O}(\left|\vec{M}\right| \times |A| \times |M_a| \times |M_a|)$, where M_a is the number of moves of some agent $a \in A$. For the worst case, we have to consider that each agent may have $\vec{M}$ moves: $\mathcal{O}(\left|\vec{M}\right| \times |A| \times \left|\vec{M}\right| \times \left|\vec{M}\right|)$. Since we are allowed to assume that the number of move vectors equals the number of directions, the length of the formula φ_{contr} is $\mathcal{O}(2^{3\times|I|} \times |A|)$.

Let us then look at the formula $\varphi_{well-formed}$ which gets repeated for every basic formula. Because of the quantification of moves of the agents, we repeat each of the requirements $\left|\vec{M}\right| \times |A|$ times. Besides the quantification, the formulas $\varphi_{\forall moves}$ and $\varphi_{\exists moves}$ have summarized the length $\mathcal{O}(|A|)$. The length of the subformula φ_{trans} is bounded by $\mathcal{O}(\left|\vec{M}\right| \times |A|)$. The complete length of $\varphi_{well-formed}$ then is: $\mathcal{O}(2^{2\times|I|} \times |A|^2)$.

Finally, we analyze the subformula $\varphi_{satisfaction}$. As the length of the subformulas $\varphi_{witness}$ and $\varphi_{cwitness}$ are both constant, the length of $\varphi_{satisfaction}$ is linear in the length of the ATL*-subformula. As we repeat this for all ATL*-subformula, the length of $\varphi' \wedge \bigwedge_{\varphi_b \in B_\varphi} \varphi_{satisfaction}(\varphi_b)$ is linear in the length of the original ATL* specification φ, hence in $\mathcal{O}(|\varphi|)$.

Combining all parts, the length of the whole formula is in $\mathcal{O}(2^{2\times|I|} + 2^{3\times|I|} \times |A| + |B_\varphi| \times (2^{|I|} \times |A|^2) + |\varphi|)$. We give an upper bound for this formula assuming the theoretical worst case: $|I| = |B_\varphi| = |A| = |\varphi|$:

$$\mathcal{O}(|\varphi| \times 2^{3\times|\varphi|}).$$

Our reduction therefore brings an exponential blow-up to the length of the formula. Considering the polynomial blow up from HyperLTL synthesis our approach still brings only one exponential blowup to the synthesis complexity.

Without any optimizations, our approach of synthesizing ATL* specifications is therefore 3-EXPTIME in the length of the ATL* specification. Considering the fact that all of the formulas we added are just disjunctions and conjunctions over atomic propositions, without temporal operators besides next and globally, it is to expect that optimizations in further work, lead to improvements. Additionally, in practice, the number of input variables might be relatively small. In fact for formulas with a number of inputs logarithmic to the length of the formula, the exponential blow-up disappears.

3.3 Correctness

Let φ^{ATL} be some arbitrary ATL* formula. Further let φ^{LTL} an LTL formula which was constructed from φ^{ATL} using the proposed reduction. We know from ➔Subsection 3.1.5 that it is sufficient to consider only explicit models realizing the formula φ^{ATL}.

➔Sec. 3.1, Page 24

Lemma 3.15. *Assume that S is an explicit model which realizes φ^{ATL}. Then there is a system S' which realizes the formula φ^{LTL}, constructed using the proposed reduction.*

Proof. We first add several propositions to the explicit model, but we will not use them for now. Of course the system is still be an explicit model of φ^{ATL}, regardless of the propositions we add.

The following propositions are added to S:

- $prop_{move}(m, a)$, which will encode the moves available to each agent at each state.
- $prop_{moveVec}(\vec{m}, m)a$, which will encode the set of all possible move vectors.

We are able to encode every move vector and move, we may get from $\vec{M}$ in the system S, since move vectors and moves are bounded by possible inputs.

Secondly, we create a mapping of the input variables I, used in the specification φ^{ATL}, to move vectors $\vec{M}$. It is not necessary to require that this mapping is a bijection, because we know that two move vectors mapped to the same input variable set, have the same semantics, hence as we will change the spanning of the explicit model later on, one direction in the new system is sufficient to project them.

In a next step, we will, within a single step, change the spanning of the system S from $\vec{M}$ to 2^I and remove $\vec{M}$ from the system (but leave the newly introduced propositions). This new system is called S'.

By construction, the system S' realizes φ^{LTL}. Let us explain, why the several parts of φ^{LTL} are satisfied. The LTL formula $\varphi_{well-formed}$ ensures the well-formedness of the new system S', which is satisfied, because S was an explicit model. The formula $\varphi_{satisfaction}$ we will use the encoded witnesses to test the satisfaction of basic subformulas. Since S is an explicit model, we know that this is satisfied as well. φ_{contr} is obviously satisfied, since we the encoded move vectors originated from actual move vectors. Consequently we know that S' realizes φ^{LTL}. $\square$

Lemma 3.16. *Assume that S' is a system which realizes the formula φ^{LTL}, constructed using the proposed reduction. Then there is some S, which is an explicit model of φ^{ATL}.*

Proof. Let us first consider the propositions we introduced in Subsection 3.2.2:

- $prop_{move}(m, a)$: As defined in ➔Definition 1, each agent has a set of moves per state. We are able to reconstruct these sets using this proposition. ➔Sec. 2.5, Page 11
- $prop_{moveVec}(\vec{m}, m)a$: Using this proposition, we are able to reconstruct the set of all possible move vectors. The formula φ_{contr}, which is required in each state, enforces that only move vectors accordingly to their definition can be encoded. Therefore it is possible to reconstruct all possible move vectors as defined in ➔Definition 1 ➔Sec. 2.5, Page 11 from this proposition.

Consequently it is possible to reconstruct the whole $\vec{M}$ of a CGS (Definition 1). Next we analyze the mapping of input variables to move vectors. We construct a new system, which is spanned by move vectors instead of a set of input variables. Of course, the way the proposition $prop_{move}(m, a)$ is constructed, it is not possible for one move to have several different input sets. If that would be possible, we could not change the spanning of the system S' to move vectors without loosing information. The system, spanned by the extracted $\vec{M}$, will be called S.

The system S realizes the formula φ^{ATL} by inverse construction. Let us analyze the various properties of an CGS: We require the system S to be well-formed, which is enforced in the system S' by the formula $\varphi_{well-formed}$ and by inverse construction this also holds for S. For realizing φ^{ATL}, the system S has to have witnesses for the satisfaction of its basic subformulas: Using the formulas $\varphi_{witness}$ and $\varphi_{cwitness}$ we are able to extract the witnesses and counter witnesses from the system S'. By inverse construction we know that these witnesses are witnesses for the formula φ^{ATL} as well. Having the set of witness strategies for φ^{ATL} and the system S, we know that S realizes φ^{ATL}. □

Theorem 3.17. *There is a system S which realizes an ATL* formula φ^{ATL}. Secondly some system S' realizes a LTL formula φ^{LTL}, which is constructed using the proposed reduction, if and only if the system S realizes φ^{ATL}.*

➔Sec. 3.1, Page 24

Proof. From ➔Subsection 3.1.5 we know, that it is sufficient to consider only explicit models realizing the formula φ^{ATL}. Then, the direction from φ^{ATL} to φ^{LTL} is shown in Lemma 3.15, whereas the other direction is shown in Lemma 3.16. □

Chapter 4

Reduction from Trees of Alternating Transition Systems

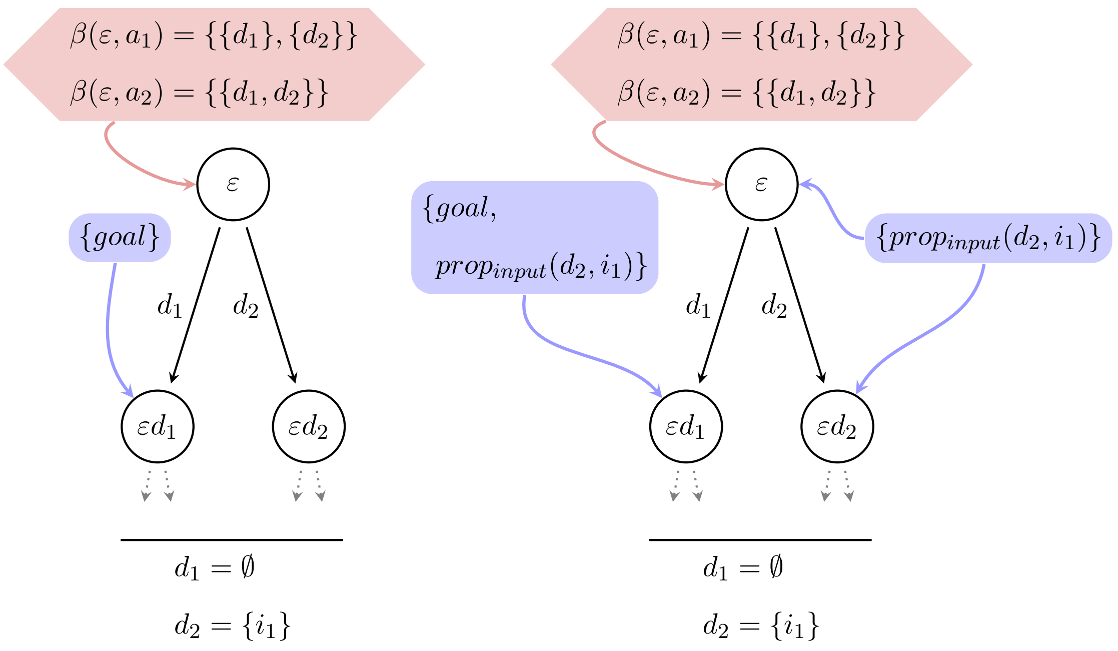
In this chapter we reduce the synthesis problem of ATL* over trees of alternating transition systems (ATS trees) to (Hyper-)LTL synthesis.

- We show how the structure of an ATS tree can be encoded completely into a computation tree, while several LTL formula guarantee the correctness.
- Secondly, we widen the tree and show how witnesses are encoded into the computation tree, while additional LTL formulas guarantee that we are only encoding semantically correct witnesses. A small HyperLTL formula guarantees the hiding of widened directions.
- Thirdly we replace parts of the ATL* formula such that we obtain a pure LTL formula which depicts in combination with the added LTL formulas and the HyperLTL formula the new (Hyper-)LTL specification.
- Finally, we show that this (Hyper-)LTL specification is realizable if and only if the ATL* specification is realizable.

4.1 Encoding the ATS Tree

In this section we encode the structure of an ATS tree into a computation tree. Since in this chapter ATL* is defined over trees of alternating transition systems, ATL* synthesis would produce such an ATS tree. LTL however is defined over a set of traces. Since a set of traces can be easily transformed into a computation tree, the goal of this section is to encode the structure of an ATS tree into a computation tree. We add several atomic propositions to the computation tree and construct multiple LTL formulas describing the relations between these propositions. Using these propositions in combination with the LTL formulas, the annotated computation tree represents an ATS tree.

- **Directions** to encode set membership of inputs in directions.
  - Formula deciding whether a direction matches the chosen input variables,


$\beta(\varepsilon, a_1) = \{\{d_1\}, \{d_2\}\}$
$\beta(\varepsilon, a_2) = \{\{d_1, d_2\}\}$
ε
$\{goal\}$
d_1
d_2
εd_1
εd_2
$d_1 = \emptyset$
$d_2 = \{i_1\}$
$\beta(\varepsilon, a_1) = \{\{d_1\}, \{d_2\}\}$
$\beta(\varepsilon, a_2) = \{\{d_1, d_2\}\}$
$\{goal, prop_{input}(d_2, i_1)\}$
ε
$\{prop_{input}(d_2, i_1)\}$
d_1
d_2
εd_1
εd_2
$d_1 = \emptyset$
$d_2 = \{i_1\}$

(a) An ATS tree. (b) An ATS tree with encoded directions.

Figure 4.1: An ATS tree with and without encoded directions

hence whether we are **following a direction**.

- **Agent behavior** to encode possible moves, in detail the function β, using set membership of directions in moves.
  - Formula deciding whether the **singleton requirement** is satisfied.

4.1.1 Encoding Directions

We introduce a proposition which indicates whether an input variable is part of a direction. This allows us to encode the structure of the ATS tree into the trace model of LTL. We are able to differentiate different branches of the ATS tree and require different conditions. A direction is, as defined in ➔Section 2.2, a set of input variables. ➔Page 7
We add this proposition to every node in the computation tree.

- $prop_{input}(d, i)$, for every direction $d \in D$ and input $i \in I$. Then

$$prop_{input}(d, i) \text{ if and only if } i \in d$$

Example 4.1 (Encoding Directions)**.** As mentioned, we introduce propositions for the directions. For the Example CGT from Figure 4.1, the added propositions are: $\{prop_{input}(d_2, i_1)\}$. For direction d_1, no proposition is introduced because no input variable is set true at this proposition. △

Following Directions

We explain how we can follow a direction in LTL. A path in the computation tree itself is characterized by the directions, hence the combinations of inputs, which hold in every node.

In the case of synthesis, we can assume that input values are set by the synthesizer. Further we can assume that the synthesizer will set each combination of input values at some point. Following the idea presented in [BSK17], we react to the input values set by the synthesizer to recognize the direction which is currently selected. While in LTL it is regularly only possible to select all traces, hence paths, this trick makes it possible to select certain paths from a computation tree using LTL and in consequence require different conditions on these paths. We call this *following a direction* or path.

We are able to decide whether we follow a certain direction d through this formula:

$$\varphi_{follow}(d) \coloneqq \bigwedge_{i \in I} prop_{input}(d, i) = i$$

Example 4.2 (Following Directions)**.** As the input variables are set by the synthesizer, we assume for this example $i_1 = true$. Accordingly to Figure 4.1 we explain how to test whether we are following a direction. For the direction d_1 we get the following:

$$\bigwedge_{i \in I} prop_{input}(d_1, i) = i$$

$$\equiv prop_{input}(d_1, i_1) = i_1 \;\equiv\; false = true \;\equiv\; false$$

Whereas for direction d_2 we get:

$$\bigwedge_{i \in I} prop_{input}(d_2, i) = i$$

$$\equiv prop_{input}(d_2, i_1) = i_1 \;\equiv\; true = true \;\equiv\; true$$

Therefore we know that we are following the direction d_2. △

4.1.2 Encoding Agent Behavior

Since the LTL setting only considers computation trees and their paths, we need to encode the additional informations of an ATS tree into a regular computation tree. We encode the function β into the nodes of the computation tree. We introduce the following boolean propositions such that for every node n:

- $prop_\beta(a, M, d)$ for all moves $M \in 2^D$, agents $a \in A$ and directions $d \in D$, such that

$$prop_\beta(a, M, d) \coloneqq [M \in \beta(a, n)] \wedge [d \in M]$$

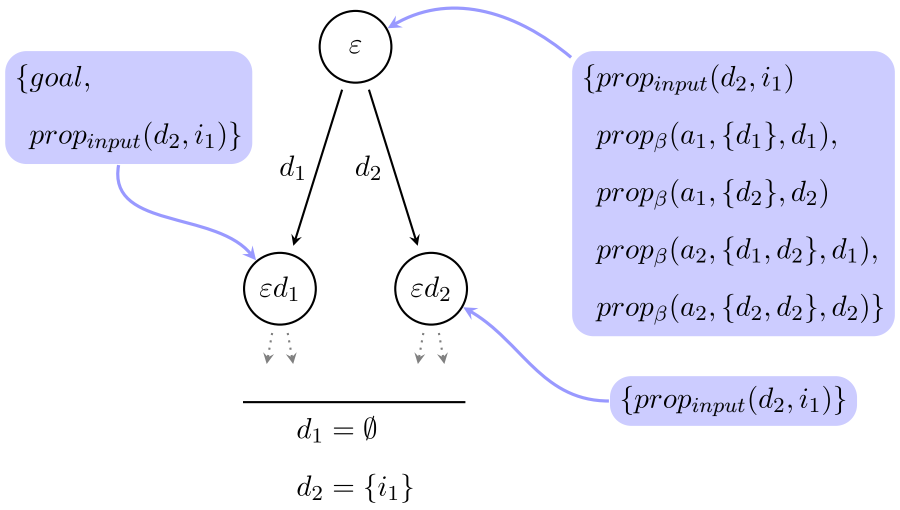

ε
$\{goal, prop_{input}(d_2, i_1)\}$
d_1
d_2
εd_1
εd_2
$\{prop_{input}(d_2, i_1)$ $prop_\beta(a_1, \{d_1\}, d_1)$, $prop_\beta(a_1, \{d_2\}, d_2)$ $prop_\beta(a_2, \{d_1, d_2\}, d_1)$, $prop_\beta(a_2, \{d_2, d_2\}, d_2)\}$
$\{prop_{input}(d_2, i_1)\}$
$d_1 = \emptyset$
$d_2 = \{i_1\}$

Figure 4.2: Example of the encoded behavior.

Example 4.3 (Encoding Agent Behavior)**.** To encode the behavior, we add the following propositions into the nodes of our example. The new available labels are:

$$\{prop_\beta(a_1, \{d_1\}, d_1), prop_\beta(a_1, \{d_1\}, d_2),$$
$$prop_\beta(a_1, \{d_1, d_2\}, d_1), prop_\beta(a_1, \{d_1, d_2\}, d_2),$$
$$\textit{the same for the agent } a_2\}$$

Depending on the actual behavior function $\beta(\varepsilon, a_1) = \{\{d_1\}, \{d_2\}\}, \beta(\varepsilon, a_2) = \{\{d_1, d_2\}\}$, we extend the running example to Figure 4.2, by adding the following propositions to the label of node ε. This will be done for all nodes n, depending on the results of $\beta(n, a_1)$ and $\beta(n, a_2)$, but for simplicity reasons we only consider the first node ε here. △

As mentioned in the definition of β, we require that the intersection of possible moves from all agents is a singleton.

Singleton Requirement

➔Sec. 2.5, Page 13

The singleton requirement is defined in ➔Definition 3. To be able to translate this requirement in LTL we rewrite the singleton requirement as following:
We assume an arbitrary but fixed n:

$$\forall M_1 \in \beta(n, a_1) \cdots \forall M_m \in \beta(n, a_m).$$
$$\exists d \in D. \forall d' \in D. (\forall M \in \{M_1 \cdots M_m\}. d' \in M) \leftrightarrow (d = d')$$

Using the subformula $\bigvee_{d \in D} prop_\beta(a_1, M, d)$, deciding whether a move M is a result of β we can construct the following LTL formula:

$$\varphi_{single} \coloneqq \Box (\bigwedge_{\substack{M_1 \in 2^D \\ \vdots \\ M_m \in 2^D}} (\bigvee_{d \in D} prop_\beta(a_1, M_1, d) \wedge \cdots \wedge \bigvee_{d \in D} prop_\beta(a_m, M_m, d) \rightarrow$$

$$\bigvee_{d \in D} \bigwedge_{d' \in D} ((\bigwedge_{i \in \{1 \cdots m\}} prop_\beta(a_i, M_i, d')) \leftrightarrow (\varphi_{d-equal}(d, d'))))$$

Since directions are formally defined as a set of input variables: $D = 2^I$, we define $\varphi_{d-equal}$ to be the comparison if d and d' contain exactly the same input variables:

$$\varphi_{d-equal}(d, d') \coloneqq \bigwedge_{i \in I} prop_{input}(d, i) \leftrightarrow prop_{input}(d', i).$$

4.2 Encoding Witnesses

In this section we encode witnesses of satisfaction into the computation tree. For that we assume that the ATS tree realizes an ATL* specification. As witnesses of satisfaction we examine strategies and plays, defined in the ATL* semantics. To make clear why we classify strategies and plays as witnesses, let us recall how strategies and plays correspond. A set of plays is made up of the strategies and capabilities of groups of agents. Capabilities originate definitely from agent behavior and therefore are part of the ATS tree, not part of the witnesses. Strategies however are part of the witnesses since their value influences the satisfaction of an ATL* formula without being part of the ATS tree. As a consequence, plays and strategies witness the satisfaction of an ATL* formula. For the labeling of witnesses, we assume that the ATS tree satisfies the ATL* specification. Therefore we know that witnesses exist, despite the fact that the witnesses are not visible in an ATS tree. Using the following propositions we make these witnesses visible by encoding them into the annotated computation tree. However, as before, to guarantee correctness, we need introduce several LTL formulas describing relations between the witnesses and other propositions. These relations originate from the definitions of plays and strategies in the ATL* semantics, thus guarantee that we are only encoding semantically correct witnesses. If the ATS tree does not realize the specification, there are no witnesses, hence we do not encode any witnesses.

- **Strategies** to encode the strategy function f_a , which is leading to path satisfaction.
  - Formula deciding whether a **strategy represents a selection of moves**.
- **Plays** to encode the set of paths $plays(n, F_{A'})$, which is leading to path satisfaction.
  - Formula deciding whether the encoding represents a set of **plays as the outcome of a strategy**.

4.2.1 Widening the Computation Tree

Before we encode the witnesses of the ATS trees into computation trees, we need to widen the computation tree. Widening the computation tree gives us the possibility to encode the necessary amount of informations into the tree. More precisely it is necessary to encode the witnesses, hence strategies and plays, without needing an infinite amount of propositions. To explain why this widening is necessary, we look at what plays and strategies formally are and how they are constructed.

A play is a path of an ATS tree, therefore the set plays describes a set of paths. We may consider this set of paths a tree, since all paths in the set plays start at the same node. The number of different set of plays is possibly unbounded because for each basic subformula a set of plays is constructed. Consider for example the formula $\Box\langle\langle a_1\rangle\rangle \bigcirc x$, where a_1 is an agent and x some atomic proposition. For each evaluation of the basic subformula $\langle\langle a_1\rangle\rangle \bigcirc x$, a set of plays needs to be constructed. As the basic subformula is required globally this leads to infinitely many sets of plays. Not only is this true for the ATS tree in general, but also for a single node. Hence it is possible that a single node is part of infinitely many sets of plays.

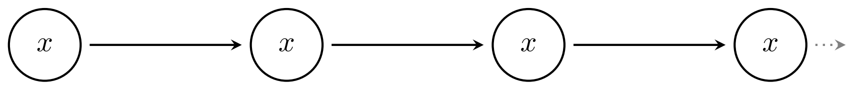
A strategy for an agent is a mapping of states to moves. Since plays are based on strategies, such that a different strategy would lead to a different set of play, we need as many strategy sets as plays. In fact it is sufficient to have exactly as many strategy sets as sets of plays, since their only purpose is to build the set of plays. For the same reason it is possible to restrict the domain of the strategy function to only such nodes, which are part of the set of plays the strategy is used to construct. However this still leaves us with an infinite amount of strategies per node.

Widening

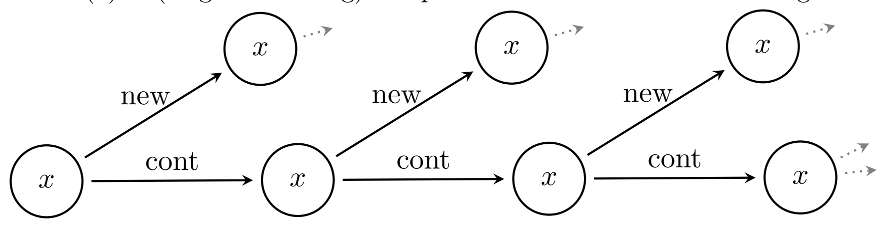
As a solution we perform a widening on the computation tree. It is not necessary to widen the ATS tree as well, as we use the annotated computation tree to encode the witnesses. Similar to [Sch08], we widen the computation tree by duplicating each direction. We call this a boolean widening. Technically this is done by inventing a fresh input variable. Let us argue why this boolean widening leads to a finite amount of strategies and plays per node and therefore makes it possible to encode arbitrary witnesses into the computation tree. A similar technique was used in ➔ Subsection 3.1.3. ➔ Sec. 3.1, Page 20

We encode plays and strategies node wise, therefore the goal is to have a finite amount of witnesses, hence strategies and sets of plays in each node.

Assume that there are only two strategies per agent and basic formula, and two sets of plays per basic formula in each node n. For an arbitrary but fixed basic formula and an agent a, let us call them f_a^{new} and f_a^{cont}, while the plays are called $plays(n_0, F_{A'}^{new})$ and $plays(n_0, F_{A'}^{cont})$. As implied, the set of plays $plays(n_0, F_{A'}^{new})$ originates from the the strategies f_a^{new} (for all a in A'), analogously for the second set of plays. Let us remark that we are only able to differentiate between these two strategies on a per node level. Generally a strategy consists of a new part and a continued part: $f_a^{new} \,\dot\cup\, f_a^{cont} = f_a$, respectively $plays(n_0, F_{A'}^{new}) \,\dot\cup\, plays(n_0, F_{A'}^{cont}) = plays(n_0, F_{A'})$.



(a) A (single-branching) computation tree before the widening



(b) The (single-branching) computation tree after the widening

Figure 4.3: A schematic example for the widening

We demonstrate a technique such that using this technique the widened tree using the bound of two is not more restricted than the original tree with no bound:

We split the directions of each node into two halves, one with directions including the fresh input variable, which was used to widen the tree and one with directions excluding the fresh input variable. We call the first half *continue directions*, the second half *new directions*. Whenever we are at the initial node n_0 of a set of plays $plays(n_0, F_{A'})$, we first go into a *new direction*, after that, hence whenever we are just traversing a node, we use the *continue direction*. Similarly we denote the part of the set of plays after going into a *new direction*: $plays(n_0, F_{A'}^{new})$, after going into a *continue direction* we denote the set $plays(n_0, F_{A'}^{cont})$. Using this technique, in each node are at most two different set of plays per basic subformula, namely $plays(n_0, F_{A'}^{new})$ and $plays(n_0, F_{A'}^{cont})$. One which is starting at that node and one which is traversing the node. Consider for Example Figure 4.3, which displays a (single-branching) computation tree in which $\langle\langle a\rangle\rangle \Box x$ holds. In Figure 4.3a, we would need to label a node in the infinite future, with infinitely many witnesses as they summarize for each node. In Figure 4.3b, however, due to the widening, by taking *new* after instancing a new witness and *cont* for continuation, only two witnesses exist in each node.

Since strategies only purpose is to build sets of plays, we proceed analogously with the strategies f_a^{new} and f_a^{cont}.

Hiding Function

To guarantee, that the widening will not falsify the system, we introduce a hiding function h. Let $D^{cont} = \{d_1, \cdots, d_m\}$ be the set of directions, we called *continue directions* and let $D^{new} = \{d_{m+1}, \cdots d_{2m}\}$ be the set of directions we called *new directions*. We hide the new input variable by replacing each $d_j \in \{d_{m+1}, \cdots, d_{2m}\}$ in a path π by the direction d_{j-m}. We call this new path π' and the hiding function is defined as $h : (D^{cont} \cup D^{new})^* \to D^*$, such that for all possible paths π, $h(\pi) = \pi'$.

The widened computation tree therefore is $(\{D^{cont} \cup D^{new}\} = 2^{I'}, L = 2^O, V, l \circ h)$, where I' is the original input variable set plus the fresh input variable.

Let us argue how the hiding function is implemented in our reduction. The hiding function does hide the *new directions* completely from the labeling function. In consequence we can not distinguish between the propositions of two nodes, which are mapped to the same node through the hiding function. In our reduction this is implemented by two parts.

- First, by simply not using the fresh input variable in all propositions we construct, we prohibit ourself to distinguish between these directions, based on the propositions.
- Secondly, we attend the problem that the LTL synthesizer still uses the new input variable for the branching, hence given two directions, which are equivalent under the hiding function, leading to two different nodes. Besides the fact that, based on the propositions, we do not know which direction was used to get to these nodes, we still are in different nodes, which may lead to a different evaluation of the formula.

Because of this, we introduce behavior equality. Two traces are behavior equal, if all propositions in each state, besides witnesses, are equal. We introduce a small HyperLTL formula, which enforces behavior equality between widened traces, hence traces, which have, up to the new input variable, the same input sequences. That way, we do not really hide the new input variable, but enforce the same behavior on traces which originated from a single trace.

Let I be the set of input variables without the newly introduced proposition. Let the set $AP \setminus witness$, be the set of all atomic proposition we introduced up to this section. $AP \setminus witness = \{prop_{input}(d, i) \mid d \in D, i \in I\} \cup \{prop_{\beta}(a, M, d) \mid a \in A, M \in 2^D, d \in D\}$

$$\varphi_{behavior-equal} = \forall \pi, \pi'. \square((\bigwedge_{i \in I} i_\pi \leftrightarrow i_{\pi'}) \rightarrow (\bigwedge_{ap \in AP \setminus witness} ap_\pi \leftrightarrow ap_{\pi'}))$$

➔Sec. 3.2, Page 28

The same formula was also used in ➔Subsection 3.2.3 from the CGS approach.

Example 4.4. As explained in Figure 4.4, we perform a boolean widening to the ATS tree from Figure 4.2. We duplicate each direction by introducing a new input variable i_2. We call the the new set of directions $\{d_1^c, d_1^n, d_2^c, d_2^n\}$, where c for *cont* are directions without the new input variable and n for *new* are directions including the new input variable. It is important that existing labeling does not include the new directions, hence that it only differs between d_1 and d_2 and does not know the input variable i_2. By using the formula $\varphi_{t-equal}$, we guarantee that propositions are simply duplicated but not changed.

△

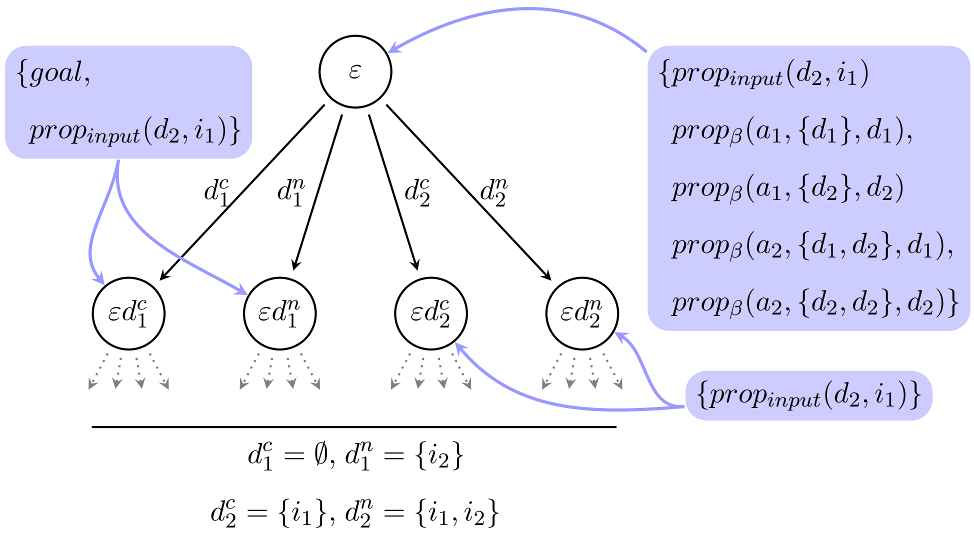

$\{goal, prop_{input}(d_2, i_1)\}$
ε
$\{prop_{input}(d_2, i_1)$
$prop_\beta(a_1, \{d_1\}, d_1),$
$prop_\beta(a_1, \{d_2\}, d_2)$
$prop_\beta(a_2, \{d_1, d_2\}, d_1),$
$prop_\beta(a_2, \{d_2, d_2\}, d_2)\}$
d_1^c
d_1^n
d_2^c
d_2^n
εd_1^c
εd_1^n
εd_2^c
εd_2^n
$\{prop_{input}(d_2, i_1)\}$
$d_1^c = \emptyset,\ d_1^n = \{i_2\}$
$d_2^c = \{i_1\},\ d_2^n = \{i_1, i_2\}$

Figure 4.4: Example of a widened ATS tree

4.2.2 Encoding Strategies

A strategy is defined in the ATL* semantics (➔Subsection 2.5.5) as a mapping from states and agents to moves M. Instead of encoding all possible strategies, we only encode those strategies, which lead to satisfaction, if possible. As we argued before in Subsection 4.2.1, it is sufficient to consider only two different arbitrary strategies per node, we denote with f_a^{new} and f_a^{cont}. ➔Sec. 2.5, Page 16

Consequently for an arbitrary but fixed basic formula φ_b, we introduce the following propositions at every node n:

- $prop_{strat}(a, d, b^*)$ where $* \in \{new, cont\}$, for all $a \in A$ and $d \in D$, such that

  $$prop_{strat}(a, d, b^{new}) \text{ if and only if } d \in f_a^{new}(n)$$

  and

  $$prop_{strat}(a, d, b^{cont}) \text{ if and only if } d \in f_a^{cont}(n)$$

Let us remark that the distinction between the two strategies f_a^{new} and f_a^{cont} is only done on a per node basis. Strategy wise a strategy contains a new and a continued part: $f_a^{new} \dot\cup f_a^{cont} = f_a$. We determine the exact pattern in ➔Subsection 4.2.3. ➔Sec. 4.2, Page 48

Example 4.5 (Encoding Strategies)**.** To encode strategies into the example ATS tree, we need to consider the ATL* formula which should hold on the example ATS tree:

$$\langle\langle\{a_1\}\rangle\rangle \bigcirc goal.$$

The set of labels will be extended with the following set of new propositions.

$$\{prop_{strat}(a_1, d_1, b^{new}_{\langle\langle a_1\rangle\rangle \bigcirc goal}), prop_{strat}(a_1, d_2, b^{new}_{\langle\langle a_1\rangle\rangle \bigcirc goal}),$$
$$prop_{strat}(a_1, d_1, b^{cont}_{\langle\langle a_1\rangle\rangle \bigcirc goal}), prop_{strat}(a_1, d_2, b^{cont}_{\langle\langle a_1\rangle\rangle \bigcirc goal}),$$
$$\textit{the same for the agent } a_2\}$$

As strategies are part of the witnesses, the satisfaction of the formula depends on the right choice of strategies. For this example, the strategy for agent a_1 to satisfy $\bigcirc goal$ is to choose the set $\{d_1\}$: $f_{a_1}(\varepsilon) = \{d_1\}$. Therefore we label node ε with the direction d_1, as pictured in Figure 4.5. For agent a_2 no strategy is necessary. We choose the *new*-instance for the strategy for reasons that will be explained in Example 4.9.

△

Relation Between Strategies and Behavior

For every node n, we need to guarantee that the local encoding of a strategy $f_a(n)$ represents a move, hence for all nodes n and basic formulas $\forall f_a.\, f_a(n) \in \beta(n, a)$.

We have to create the following formula to be able to translate this to LTL: For some agent and some basic formula we require in each state for each strategy, hence for f_a^{new} and f_a^{cont}:

$$\exists M \in 2^D.\, \forall d \in D.\, (d \in f_a(n) \leftrightarrow (d \in M \wedge M \in \beta(a, n))).$$

The LTL formula per basic formula and instance: b^* ($* \in \{new, cont\}$) and agent a for the relation between a strategy and the behavior then is:

$$\varphi_{strategy}(a, b) := \square\,($$
$$(\bigvee_{M\in 2^D} \bigwedge_{d\in D} (prop_{strat}(a, d, b^{new}) \leftrightarrow prop_\beta(a, M, d))\wedge$$
$$(\bigvee_{M\in 2^D} \bigwedge_{d\in D} (prop_{strat}(a, d, b^{cont}) \leftrightarrow prop_\beta(a, M, d))$$

Example 4.6. In this example we test if the encoded strategy in Figure 4.5 matches the behavior accordingly to the formula $\varphi_{strategy}$. Adjusted to our example, the formula $\varphi_{strategy}$ for agent a_1 and basic formula $\langle\langle a_1\rangle\rangle \bigcirc goal$ becomes

$$
\begin{aligned}
&\Box((\\
&\quad ((prop_{strat}(a_1, d_1, b^{new}_{\langle\langle a_1\rangle\rangle \bigcirc goal}) \leftrightarrow prop_\beta(a_1, \{d_1\}, d_1))\wedge\\
&\quad\quad (prop_{strat}(a_1, d_2, b^{new}_{\langle\langle a_1\rangle\rangle \bigcirc goal}) \leftrightarrow prop_\beta(a_1, \{d_1\}, d_2)))\vee\\
&\quad ((prop_{strat}(a_1, d_1, b^{new}_{\langle\langle a_1\rangle\rangle \bigcirc goal}) \leftrightarrow prop_\beta(a_1, \{d_2\}, d_1))\wedge\\
&\quad\quad (prop_{strat}(a_1, d_2, b^{new}_{\langle\langle a_1\rangle\rangle \bigcirc goal}) \leftrightarrow prop_\beta(a_1, \{d_2\}, d_2)))\vee\\
&\quad ((prop_{strat}(a_1, d_1, b^{new}_{\langle\langle a_1\rangle\rangle \bigcirc goal}) \leftrightarrow prop_\beta(a_1, \{d_1, d_2\}, d_1))\wedge\\
&\quad\quad (prop_{strat}(a_1, d_2, b^{new}_{\langle\langle a_1\rangle\rangle \bigcirc goal}) \leftrightarrow prop_\beta(a_1, \{d_1, d_2\}, d_2)))\\
&) \wedge (\\
&\quad ((prop_{strat}(a_1, d_1, b^{cont}_{\langle\langle a_1\rangle\rangle \bigcirc goal}) \leftrightarrow prop_\beta(a_1, \{d_1\}, d_1))\wedge\\
&\quad\quad (prop_{strat}(a_1, d_2, b^{cont}_{\langle\langle a_1\rangle\rangle \bigcirc goal}) \leftrightarrow prop_\beta(a_1, \{d_1\}, d_2)))\vee\\
&\quad ((prop_{strat}(a_1, d_1, b^{cont}_{\langle\langle a_1\rangle\rangle \bigcirc goal}) \leftrightarrow prop_\beta(a_1, \{d_2\}, d_1))\wedge\\
&\quad\quad (prop_{strat}(a_1, d_2, b^{cont}_{\langle\langle a_1\rangle\rangle \bigcirc goal}) \leftrightarrow prop_\beta(a_1, \{d_2\}, d_2)))\vee\\
&\quad ((prop_{strat}(a_1, d_1, b^{cont}_{\langle\langle a_1\rangle\rangle \bigcirc goal}) \leftrightarrow prop_\beta(a_1, \{d_1, d_2\}, d_1))\wedge\\
&\quad\quad (prop_{strat}(a_1, d_2, b^{cont}_{\langle\langle a_1\rangle\rangle \bigcirc goal}) \leftrightarrow prop_\beta(a_1, \{d_1, d_2\}, d_2)))))
\end{aligned}
$$

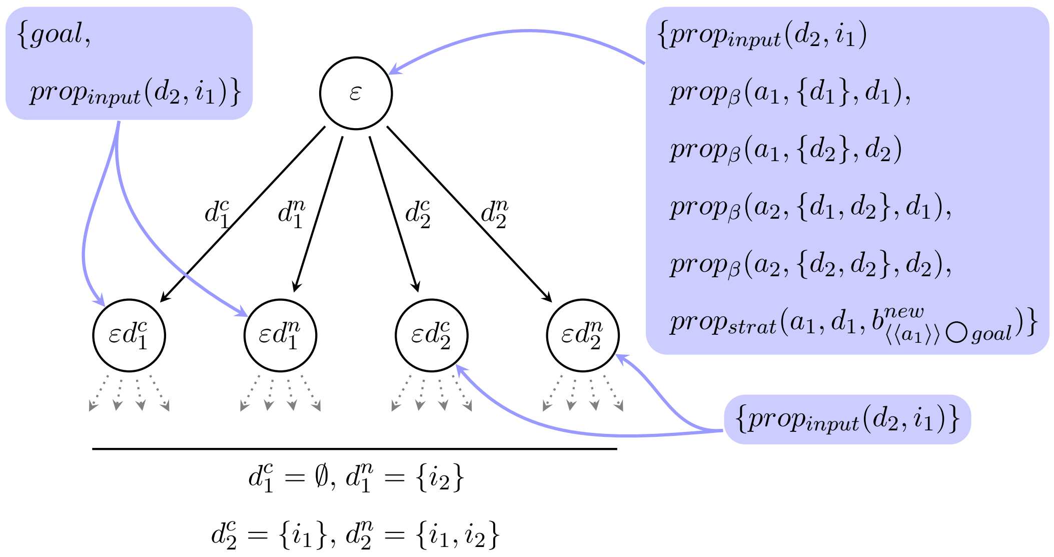

$\{goal,$
$prop_{input}(d_2, i_1)\}$
ε
d_1^c
d_1^n
d_2^c
d_2^n
εd_1^c
εd_1^n
εd_2^c
εd_2^n
$\{prop_{input}(d_2, i_1)$
$prop_\beta(a_1, \{d_1\}, d_1),$
$prop_\beta(a_1, \{d_2\}, d_2)$
$prop_\beta(a_2, \{d_1, d_2\}, d_1),$
$prop_\beta(a_2, \{d_2, d_2\}, d_2),$
$prop_{strat}(a_1, d_1, b^{new}_{\langle\langle a_1\rangle\rangle \bigcirc goal})\}$
$\{prop_{input}(d_2, i_1)\}$
$d_1^c = \emptyset,\ d_1^n = \{i_2\}$
$d_2^c = \{i_1\},\ d_2^n = \{i_1, i_2\}$

Figure 4.5: Example of encoded strategies

As one can test for the strategy from Figure 4.5, this formula holds. Additionally this formula does not hold if the stated strategy would have been the strategy of agent a_2, hence the node ε would have been labeled with $prop_{strat}(a_2, d_1, b^{new}_{\langle\langle a_1\rangle\rangle \bigcirc goal})$. △

4.2.3 Encoding Plays

Next we need to tag the set of paths on which the path formula of an ATL* specification should hold. As defined in the ATL* semantics (➔Subsection 2.5.5), the path formula has to hold on all paths, which are members in the set $plays(n_0, F_{A'})$, for some n_0 and coalition A'. ➔Sec. 2.5, Page 16

As we argued in ➔Subsection 4.2.1, we know that it is adequate to consider only two different set of plays per node, which we label with *new* and *cont*. ➔Sec. 4.2, Page 42

We first fix an arbitrary basic subformula and introduce in every node n the following boolean propositions. We differ between the two sets of plays, starting at the node n: $plays(n, F^{new}_{A'})$ and the set of plays going through the node n: $plays(n_0, F^{cont}_{A'})$, where n_0 is some initial node of the set plays.

- $prop_{plays}(b^{new})$, such that

$$prop_{plays}(b^{new}) \text{ if and only if } n = n_0 \land \exists \pi = n_0 \cdots \in plays(n_0, F^{new}_{A'})$$

- $prop_{plays}(b^{cont})$, such that

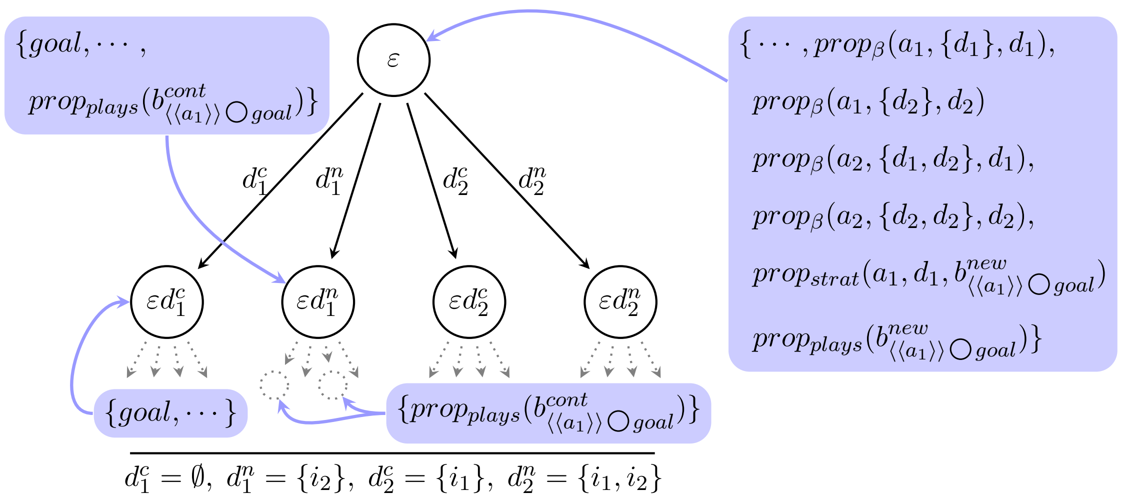
$$prop_{plays}(b^{cont}) \text{ if and only if } n = n_0 \cdots d \land \exists \pi = n_0 \cdots d \cdots \in plays(n_0, F^{cont}_{A'})$$

➔Sec. 4.1, Page 38 As before in ➔Subsection 4.1.1, we are considering a per node view. Plays wise a set plays contains a new part and a continued part: $plays(n_0, F^{new}_{A'}) \dot\cup plays(n_0, F^{cont}_{A'}) = plays(n_0, F_{A'})$, where the node n_0 belongs to $plays(n_0, F^{new}_{A'})$ and all other node which are part of $plays(n_0, F_{A'})$ are assigned to $plays(n_0, F^{cont}_{A'})$.

Example 4.7 (Encoding Plays)**.** For the running example we extend the labels with the following two propositions.

$$\{prop_{plays}(b^{new}_{\langle\langle a_1\rangle\rangle \bigcirc goal}), prop_{plays}(b^{cont}_{\langle\langle a_1\rangle\rangle \bigcirc goal})\}$$

We encode this into the computation tree by extending the labeling function (Figure 4.6). As plays are witnesses, the satisfaction of the formula depends on the right labeling. We label the root node with the start of the play and node εd^n_1 with the continuation. For this example, we assume that the play has a valid continuation after εd^n_1, which is not displayed in the figure. △



$\{goal, \cdots,$
$prop_{plays}(b^{cont}_{\langle\langle a_1\rangle\rangle \bigcirc goal})\}$
ε
d_1^c
d_1^n
d_2^c
d_2^n
εd_1^c
εd_1^n
εd_2^c
εd_2^n
$\{\cdots, prop_\beta(a_1, \{d_1\}, d_1),$
$prop_\beta(a_1, \{d_2\}, d_2)$
$prop_\beta(a_2, \{d_1, d_2\}, d_1),$
$prop_\beta(a_2, \{d_2, d_2\}, d_2),$
$prop_{strat}(a_1, d_1, b^{new}_{\langle\langle a_1\rangle\rangle \bigcirc goal})$
$prop_{plays}(b^{new}_{\langle\langle a_1\rangle\rangle \bigcirc goal})\}$
$\{goal, \cdots\}$
$\{prop_{plays}(b^{cont}_{\langle\langle a_1\rangle\rangle \bigcirc goal})\}$
$d_1^c = \emptyset,\ d_1^n = \{i_2\},\ d_2^c = \{i_1\},\ d_2^n = \{i_1, i_2\}$

Figure 4.6: Example of encoded plays

Plays Recognition

After adding propositions to the computation tree, which describe the sets of plays, we construct a LTL formula which recognizes a path, which is part of a set plays. As mentioned before $plays(n_0, F^{new}_{A'}) \dot\cup plays(n_0, F^{cont}_{A'}) = plays(n_0, F_{A'})$, hence some nodes of this set belong to the new part and some belong to the continued part. A path in plays always starts with n_0 for a set of plays $plays(n_0, F_{A'})$, hence the node n_0 is part of the set $plays(n_0, F^{new}_{A'})$. Therefore we require the first node of the path we test, to be labeled with $prop_{plays}(b^{new})$ (for an arbitrary basic formula φ_b). All further nodes are required to be labeled with $prop_{plays}(b^{cont})$ as they are part of $plays(n_0, F^{cont}_{A'})$.

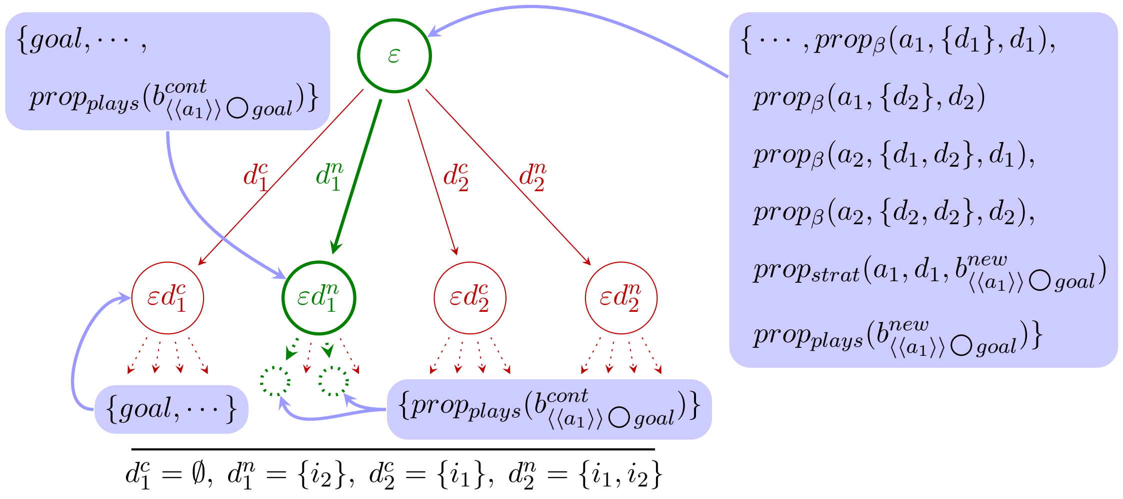
The following LTL formula implements this test for an arbitrary basic formula φ_b:

$$\varphi_{inplays}(b) := prop_{plays}(b^{new}) \wedge \bigcirc\Box prop_{plays}(b^{cont}).$$

Example 4.8. In this example, we look at the formula, which allows us to recognize plays in the ATS tree. Starting at the root node, it requires a $prop_{plays}(b^{new})$ proposition, followed by $prop_{plays}(b^{cont})$ propositions in all further states of a path. In Figure 4.7, the thick green paths are paths on which the formula holds, whereas on red thin paths, the formula does not hold. As before, we assumed a valid continuation of the system after the first transitions. △

Relation Between Plays, Strategies and Behavior

Certainly there is a relation between plays, strategies and agent behavior, as plays are defined as the outcome of a strategy. As before, we are fixing one arbitrary basic subformula first. The relations result from the ATL* semantics in ➔Subsection 2.5.5, especially ➔Sec. 2.5, Page 16



$\{goal, \cdots,$
$prop_{plays}(b^{cont}_{\langle\langle a_1\rangle\rangle \bigcirc goal})\}$
ε
d_1^c
d_1^n
d_2^c
d_2^n
εd_1^c
εd_1^n
εd_2^c
εd_2^n
$\{goal, \cdots\}$
$\{prop_{plays}(b^{cont}_{\langle\langle a_1\rangle\rangle \bigcirc goal})\}$
$\{\cdots, prop_\beta(a_1, \{d_1\}, d_1),$
$prop_\beta(a_1, \{d_2\}, d_2)$
$prop_\beta(a_2, \{d_1, d_2\}, d_1),$
$prop_\beta(a_2, \{d_2, d_2\}, d_2),$
$prop_{strat}(a_1, d_1, b^{new}_{\langle\langle a_1\rangle\rangle \bigcirc goal})$
$prop_{plays}(b^{new}_{\langle\langle a_1\rangle\rangle \bigcirc goal})\}$
$d_1^c = \emptyset,\ d_1^n = \{i_2\},\ d_2^c = \{i_1\},\ d_2^n = \{i_1, i_2\}$

Figure 4.7: Example of encoded plays with marked plays

the definition of the set $plays(n, F_{A'})$.

For this relation we consider a per strategy view as this is necessary to describe the set $plays(n, F_{A'})$ as a whole. As defined, each strategy f_a is projected on the computation tree through the propositions $prop_{strat}(a, d, b^{new})$ and $prop_{strat}(a, d, b^{cont})$ (for some $d \in D$). We require that exactly one node per strategy belongs to f_a^{new} and all other nodes of the strategy belong to f_a^{cont}. Further we require that such nodes belonging to f_a^{new} also belong to $plays(n, F_{A'}^{new})$, where $F_{A'}^{new}$ is the aggregation of f_a^{new} for the agents a in A'. Analogously for f_a^{cont}. In the following we refine the definition of a set plays, such that it differentiates between f_a^{new} and f_a^{cont}.

$$plays(n, F_{A'}) = \{nd_1 \cdots \mid d_1 \in \mathcal{C}^{A\setminus A'}_{n\cdots d_{i-1}} \wedge \forall f_a^{new} \in F_{A'}^{new}.\, d_1 \in f_a^{new}(n) \wedge$$
$$\forall i \geq 2.\, d_i \in \mathcal{C}^{A\setminus A'}_{n\cdots d_{i-1}} \wedge \forall f_a^{cont} \in F_{A'}^{cont}.\, d_i \in f_a^{cont}(n \cdots d_{i-1})\}$$

We rewrite this definition such that we are able to translate this into LTL:

Let $\pi = nd_1 \cdots$, then

$$\pi \in plays(n, F_{A'}) \leftrightarrow (d_1 \in \mathcal{C}^{A\setminus A'}_{n\cdots d_{i-1}} \wedge \forall f_a^{new} \in F_{A'}^{new}.\, d_1 \in f_a^{new}(n) \wedge$$
$$\forall i \geq 2.\, d_i \in \mathcal{C}^{A\setminus A'}_{n\cdots d_i} \wedge \forall f_a^{cont} \in F_{A'}^{cont}.\, d_i \in f_a^{cont}(n \cdots d_{i-1})))$$

Let us consider right and left part of the equivalence separately. The left part is already covered by the formula $\varphi_{inplays}$. On the right part, we need the formula to hold for every node. In LTL we require a *globally* ($\Box$) for the second line. Furthermore we need a formula deciding whether $d_i \in \mathcal{C}^{A\setminus A'}_{n\cdots d_{i-1}}$ and $\forall f_a \in F_{A'}.\, d_i \in f_a(n)$. Since the strategy

$F_{A'}$ is a compound of agent strategies and therefore is encoded through the proposition $prop_{strat}(a, d, b^m)$ for all $a \in A$ in every single node, we are able to decide the membership through a boolean formula. The same holds for the set $\mathcal{C}^{A \setminus A'}_{n \cdots d_i}$ because it is based on agent behavior which is encoded locally through the propositions $prop_\beta(a, M, d)$ for all $a \in A$. We do this step by step:

- First we look at the formula $\forall f_a \in F_{A'}.\, d \in f_a(n)$, for any coalition A', a strategy set $F_{A'}$ and some direction d. We can simplify this formula by quantifying directly over the agents of the strategy set $F_{A'}$. $\forall a \in A'.\, d \in f_a$. We construct the following boolean formula: For some basic subformula φ_b and their identifier b_* where $* \in \{new, cont\}$ at some arbitrary node:

$$\varphi_{d \in F_{A'}}(d, b^*) := (\bigwedge_{a \in A'} prop_{strat}(a, d, b^*))$$

- Secondly, we analyze whether a direction is part of the capabilities of an agent: $d \in \mathcal{C}^a_n$, where $d \in D, a \in A$ at some arbitrary node becomes the following boolean formula:

$$\varphi_{d \in \mathcal{C}^a}(d) := (\bigvee_{M \in 2^D} prop_\beta(a, M, d))$$

- Lastly we use the second formula to aggregate over the set of agents A': $d \in \mathcal{C}^{A'}_n$, where $d \in D, A' \subseteq A$ at some arbitrary node becomes the boolean formula:

$$\varphi_{d \in \mathcal{C}^{A'}}(d) := \bigwedge_{a \in A'} \varphi_{d \in \mathcal{C}^a}(d)$$

Using the above formulas we can build the following LTL formula for the definition of plays. We introduce the notation $agents(b)$ to extract the coalition of a basic subformula b. Let $A' = agents(b)$.

$$\begin{aligned}
&\varphi_{plays}(b) := \\
&\varphi_{inplays}(b) \leftrightarrow \Big(\bigvee_{d \in D} \varphi_{follow}(d) \wedge i_{new} \wedge \varphi_{d \in \mathcal{C}^{A \setminus A'}}(d) \wedge \varphi_{d \in F_{A'}}(d, b^{new}) \wedge \\
&\qquad (\bigcirc \square \bigvee_{d \in D} \varphi_{follow}(d) \wedge \neg i_{new} \wedge \varphi_{d \in \mathcal{C}^{A \setminus A'}}(d) \wedge \varphi_{d \in F_{A'}}(d, b^{cont})) \Big)
\end{aligned}$$

This formula is the direct result of combining the constructed subformula $\varphi_{inplays}$, $\varphi_{d \in F_{A'}}$ and $\varphi_{d \in \mathcal{C}^{A'}}$ with the modified definition of plays from above. Also, we fix the branch that has to be labeled concerning *new* and *continue* directions as described in ➔Subsection 4.2.1 ➔Sec. 4.2, Page 42

Example 4.9. Let us check if these conditions hold for the running example. We already marked the paths on which $\varphi_{inplays}$ holds in Figure 4.7. As in ➔Example 4.2, we have ➔Sec. 4.1, Page 39

to assume the input variables a synthesizer might have chosen.

Let us first assume that variable i_1 is chosen to be true as well as i_2. Regarding our example that means that the path starting with $\varepsilon \cdot d_2^n \cdots$ is chosen. As visualized in Figure 4.7, this path does not satisfy the formula $\varphi_{inplays}$. To find out the direction which is used, we need to evaluate the formula φ_{follow}. As we already explained this in ➔Example 4.2, we skip this and use d_2 directly. Keep in mind that the formula φ_{follow} cannot differentiate between d_2^c and d_2^n and therefore treats it as a single direction. Since the reason, we introduced the widening was to be able to encode more witnesses than before, we need to make the distinction between direction d_2^n and d_2^c. Following the technique from Subsection 4.4, in the first step we need $i_new = i_2$ to be true, which is the case in this example. In a second step we check whether d_2 is a direction which agent a_2 is *capable* of doing, hence whether $\bigvee_{M \in 2^D}(prop_\beta(a_2, M, d_2))$ at node ε. As one can see in $Figure$ 4.7, this is true. Further we test, whether d_2 is in the strategy of a_1, hence whether ε, is labeled with $prop_{strat}(a_1, d_2, b^{new}_{\langle\langle a_1\rangle\rangle \bigcirc goal})$, which is not the case. Since the selected path does not satisfy $\varphi_{inplays}$ as well, for the selected path, this φ_{plays} holds.

➔Sec. 4.1, Page 39

As a second example, let us check whether it holds if both input variables are set false. According to φ_{follow}, we selected the path $\varepsilon d_1^c \cdots$. From Figure 4.7, we know that we are not in plays with this formula. For the right part of the equivalence, we directly notice that $i_{new} = i_2$ is not set, hence it also can not be true. As a consequence, for this path, the formula φ_{plays} does hold again.

Finally, we assume the input sequence $\{i_2\} \cdot \emptyset$. As one can see in Figure 4.7, this is a path which is marked, hence it satisfies $\varphi_{inplays}$. Further, as shown in ➔Example 4.2, we follow d_1 in the first step. Also $i_new = i_2$ is set and d_2 is a direction, a_2 is capable doing. Next we require that d_2 is in the strategy of the coalition, hence the strategy of a_1 and also it has to be a *new* instance. As you can see, the root node is labeled with $prop_{strat}(a_1, d_1, b^{new}_{\langle\langle a_1\rangle\rangle \bigcirc goal})$, hence this is true. For all next states, we only consider the direct successor, we want the same to hold but with *cont* and $\neg i_{new}$. As implied, this is also the case for this path. △

➔Sec. 4.1, Page 39

4.3 Transforming the ATL* specification

Finally, we construct a LTL formula which decides which paths of the annotated computation tree should **satisfy the path formulas** of an ATL* specification. We use this formula to replace the path quantifier of the ATL* specification and combine all the formulas to a pure LTL specification. The annotated computation tree then realizes the LTL specification if and only if an ATS tree realizes the ATL* specification.

4.3.1 Path Satisfaction

We analyze the semantics of path quantifiers in ATL* and replace the path quantification by an LTL formula. We make use of the witnesses strategies and plays, to select the paths on which the path formula has to hold. Selecting certain paths becomes possible by using the formula φ_{follow} from ➔Subsection 4.1.1.

➔Sec. 4.1, Page 38

The semantics of a basic formula in LTL is defined as:

$$n \models \langle\langle A' \rangle\rangle \psi \text{ if and only if } \exists F_{A'}. \forall \pi \in plays(n, F_{A'}).\, \pi \models \psi$$

Let φ be an ATL* formula and φ_b a basic subformula on the highest level of φ, where b indicates the subformula uniquely. Given such a subformula φ_b, we replace each basic subformula thereof $\varphi_{b^{sub}} \in \varphi_b$ with the proposition $prop_{plays}(b^{sub}_{new})$. The modified body of φ_b is called φ'_b. This leads to the following LTL formula:

$$\varphi_{satisfaction}(\varphi_b) :=\Box \left(\varphi_{inplays}(b) \leftrightarrow \varphi'_b \right),$$

Let us analyze this formula: Since path recognition for witnesses is done by the formula φ_{plays}, we replace $\exists F_{A'}.\, \pi \in plays(n, F_{A'})$ with φ_{plays}. Further, in LTL, the satisfaction of a path formula ($\pi \models \psi$) is simply the path formula itself. Therefore we replace $\pi \models \psi$ with φ'_b. The formula $\varphi_{inplays}$ will only be true, if its starts with $prop_{plays}(b^{new})$, hence assuming that $prop_{plays}(b^{new})$ is always set at the right positions, we can require this globally.

Example 4.10 (Path Satisfaction). Adapted to the running example, we know that the formula φ'_b is $\bigcirc goal$. Also we know on which paths $\varphi_{inplays}$ holds. Since only the root node is labeled with the start of plays ($prop_{plays}(b^{new}_{\langle\langle a_1 \rangle\rangle \bigcirc goal})$), we can infer from ➔Figure 4.7, that this formula is globally satisfied. △ ➔Sec. 4.2, Page 50

4.3.2 Complete LTL Formula

Finally, we combine all the LTL formulas to one complete LTL specification.

Let φ be some ATL* formula, let B_φ be the set of all basic subformulas in the formula φ. We denote a basic subformula as $\varphi_b \in B_\varphi$, where b is a unique identifier for this subformula. As mentioned before, given an ATL* formula φ we can replace all basic subformulas φ_b of φ with the proposition $prop_{plays}(b^{new})$. We denote the modified formula φ'.

$$\varphi_{behavior-equal} \land \varphi_{single} \land \varphi' \land \bigwedge_{\forall \varphi_b \in B_\varphi} \bigwedge_{a \in A} \varphi_{strategy}(a, b) \land \varphi_{plays}(b) \land \varphi_{satisfaction}(\varphi_b)$$

Let us explain how this formula works. First we require the trace equality, we mentioned in Subsection 4.2.1. Secondly, we require that the singleton requirement holds. Next, we take the original ATL* specification φ and remove all basic subformulas of the highest level b and replace them with $prop_{plays}(b^{new})$. This will ensure later on that $\varphi_{satisfaction}$ is only evaluated at the right nodes. The following part has to hold for all basic formulas and agents. First we require $\varphi_{strategy}$, which is ensuring that all labeled strategies are semantically allowed. Next we require φ_{plays} to hold, which is ensuring that all labeled sets of plays are semantically correct. Finally, we require $\varphi_{satisfaction}$ to hold. This requires that all bodies of basic subformulas are be satisfied at the correct nodes. The formula φ_{plays} of the formula $\varphi_{satisfaction}$ will only hold on the nodes labeled with $prop_{plays}(b^{new})$ and since in φ' we replaced all basic subformulas φ_b with the proposition $prop_{plays}(b^{new})$, all the basic subformulas of the highest level are ensured to hold due to the $\varphi_{satisfaction}$. Further in $\varphi_{satisfaction}$, we replaced each basic subformula of this subformula again, therefore this holds recursively.

4.3.3 Complexity of the Constructed Formula

We analyze the complexity of the new formula in comparison to the ATL* formula.

Let us first look at the HyperLTL formula $\varphi_{behavior-equal}$. For this kind of formula the synthesis of HyperLTL has a quadratic blowup [Fin+18] in comparison to LTL synthesis. Let us analyze the length of this formula. The length is dependent on the number of propositions we compare. For $prop_{input}(d, i)$, we get for all combinations of directions d and inputs i, $\mathcal{O}(|I| \times 2^{|I|})$ propositions to compare. For $prop_{\beta}(a, M, d)$, we get for all combinations of set of moves M, directions d and agents a, $\mathcal{O}(2^{|I|} \times |A| \times 2^{2^{|I|}})$ propositions. For the whole formula we are in $\mathcal{O}(2^{|I|+2^{|I|}} \times |A|)$ propositions, hence the length of the formula.

Secondly, we analyze the subformula φ_{single}. The whole formula gets repeated for each combination of moves, hence $\mathcal{O}(|A| \times 2^{2^{|I|}})$ times. Te first line then has a length of $\mathcal{O}(|A| \times 2^{|I|})$. Based on the con- and disjunction of directions in the second line, it has a length in $\mathcal{O}(2^{2\times|I|} \times |A| \times |I|)$. Overall the formula φ_{single} has a length in $\mathcal{O}(|A|^2 \times 2^{2\times|I|+2^{|I|}})$.

Let us then look at the formula $\varphi_{strategy}$ which gets repeated for every basic formula. Because of the quantification of moves, we repeat the formula $\mathcal{O}(2^{2^{|I|}})$ times. The formula itself is two times $\mathcal{O}(2^{|I|})$. The complete length of $\varphi_{strategy}$ then is: $\mathcal{O}(2^{|I|+2^{|I|}})$.

Finally, we analyze the subformula $\varphi_{satisfaction}$. As the length of the subformulas $\varphi_{inplays}$ is constant, the length of $\varphi_{satisfaction}$ is linear in the length of the ATL*-subformula. As we repeat this for all ATL*-subformula, the length of $\varphi' \wedge \bigwedge_{\varphi_b \in B_\varphi} \varphi_{satisfaction}(\varphi_b)$ is linear in the length of the original ATL* specification φ, hence in $\mathcal{O}(|\varphi|)$.

Combining all parts, the length of the whole formula is in $\mathcal{O}(|A|^2 \times 2^{2\times|I|+2^{|I|}} + 2^{|I|+2^{|I|}} + |\varphi|)$. We give an upper bound for this formula assuming the theoretical worst case: $|I| = |B_\varphi| = |A| = |\varphi|$:

$$\mathcal{O}(2^{2\times|\varphi|+2^{|\varphi|}})$$

Our reduction therefore brings an exponential blow-up to the length of the formula. Considering the polynomial blow up from HyperLTL synthesis our approach still brings only one exponential blowup to the synthesis complexity.

4.4 Correctness

Let φ^{ATL} be some arbitrary ATL* formula. Further let φ^{LTL} an arbitrary LTL formula which was constructed from φ^{ATL} using the proposed reduction.

Lemma 4.11. *Assume that S is a system which realizes φ^{ATL}. Then there is a system S' which realizes the formula φ^{LTL}, constructed using the proposed reduction.*

Proof. Let w.l.o.g. the system S be an ATS tree (➔Definition 3). Let us first encode the behavior function of ATS trees as propositions. It is clear that after encoding, the system S is still a model of φ^{ATL} since we only added proposition but did not use them for now. Also, the way the encoding is done and because agents and directions are finite, it is feasible to encode each possible behavior as proposition. Secondly, we remove the behavior function from the ATS tree and consult the new propositions for extracting behavior. The resulting system is called S_{comp}. Because each behavior can be encoded uniquely, consulting the proposition instead of the behavior function, the system preserves the model property of S, hence S_{comp} realizes φ^{ATL}. The system is called S_{comp} because it already is a computation tree. ➔Sec. 2.5, Page 13

As we argued in ➔Subsection 4.2.1, it is feasible to encode all possible strategies and their resulting plays. Since S_{comp} realizes φ^{ATL}, for each basic subformula, there are strategies and plays leading to path satisfaction. Further, all requirements we put on the propositions for strategies and plays are a direct translation of parts of the ATL* semantics. Based on these facts, it is possible to encode the strategies and plays, which will comply with the formula $\varphi_{satisfaction}$ into the system S_{comp}. We call the system resulted from S_{comp} by encoding strategies and plays S'. Consequently, by construction we know that S' realizes the formula φ^{LTL}. ➔Sec. 4.2, Page 42

□

Lemma 4.12. *Assume that S' is a system which realizes the formula φ^{LTL}, constructed using the proposed reduction. Then there is some S, which is a model of φ^{ATL}.*

Proof. As a first step we extract the behavior function β from the system S'. For each set of directions M, agent a and direction d, we test whether $prop_\beta(a, M, d)$ is true. If this is the case, we know that $d \in \beta(a, n)$ and $d \in M$. Therefore we are able to reconstruct β. The formula φ_{Single} guarantees that the extracted behavior function also satisfies the singleton requirement. We call the system with the new behavior function and without the behavior proposition S_{beh}. By inverse construction, this system already realizes the formula φ^{ATL}.

Further we argue why the system S_{beh} does in fact realize φ^{ATL} and why we can remove all other additional propositions. Consider each basic subformula b. The formula $\varphi_{satisfaction}$ guarantees, that whenever φ_{plays} holds, the body of the basic subformula is satisfied. Since existence of strategies is ensured by the formula $\varphi_{strategy}$, this is exactly the semantics definition of the satisfaction of basic subformulas in ATL*. Let us therefore analyze what φ_{plays} exactly is. The formula φ_{plays} recognizes which paths are following the definition of *plays*, where strategies $(F_{A'})$ are extracted from the propositions $prop_{strat}(a, d, b^{new})$ or $prop_{strat}(a, d, b^{cont})$. Since the number of strategies are not bounded in an ATS tree and the formula $\varphi_{strategy}$ ensures that the propositions $prop_{strat}(a, d, b^{new})$ or $prop_{strat}(a, d, b^{cont})$ follow the definition of a strategy in an ATS tree, we can extract the strategies for an ATS tree from these propositions. The same holds for *plays* and the propositions $prop_{plays}(b^{new})$ and $prop_{plays}(b^{cont})$. Therefore by extracting *plays* from the system S_{beh}, we know that these paths are, following the definition from the ATS tree, in *plays*. Therefore we know that the system S_{beh} has plays, indicating the satisfaction of a basic subformula. Consequently, doing this for every basic subformula, the system S_{beh} does realize φ^{ATL}.

Moreover, we argue that we can transform the system S_{beh} into the system S and that S also realizes φ^{ATL}. This is done by simply removing all additional propositions from S_{beh} which gives us an ATS tree. Since all propositions we removed in this step are not used by the specification φ^{ATL}, the system S realizes φ^{ATL}. □

Theorem 4.13. *There is a system S which realizes an ATL* formula φ^{ATL}. Secondly, some system S' realizes a LTL formula φ^{LTL}, which is constructed using the proposed reduction, if and only if the system S realizes φ^{ATL}.*

Proof. The direction from φ^{ATL} to φ^{LTL} is shown in Lemma 4.11, whereas the other direction is shown in Lemma 4.12. □

Chapter 5

Towards Synthesizing Smart Contracts

In this chapter we will analyze non-repudiation properties and how they can be formulated in ATL. The presented findings and results are inspired by [KR03]. Further, we will give an outlook on what needs to be considered to synthesize smart contracts and what other interesting properties for smart contracts are.

5.1 Non-Repudiation

We denote as fair non-repudiation the combination of two properties, the non-repudiation of origin (NRO) and the non-repudiation of receipt (NRR). Consider, for example, that Alice sends a message to Bob. Two scenarios may happen: Alice may deny having the sent message and Bob may deny having the message received. An illustration of the non-repudiation property can be found in Figure 5.1. Using a protocol ensuring NRO and NRR, in case of a dispute between Alice and Bob, evidences of receipt and send can be presented, either by the parties themselves or by a trusted third party. A trusted third party (TTP) is some authority, which both agree to trust. In real contracts, a notary or a bank may represent such a trusted third party. The major problem of non-repudiation protocols is ensuring fairness between these parties, hence that Alice gets an evidence of receipt if and only if Bob also gets some evidence of origin. Dealing with unreliable channels as in the Internet, complicates this even more.

Non-repudiation protocols can be classified as exchange protocols, which significantly differ from authentication or secrecy protocols. In authentication and secrecy protocols the goal is to defend against an external intruder interfering with the messages sent between participants. The participants are generally considered honest. In exchange protocols, however the presence of an intruder is not important as we consider the other participants as adversaries. Based on that authenticity or secrecy protocols have an almost linear run, while in exchange protocols we do not trust the participants to ex-

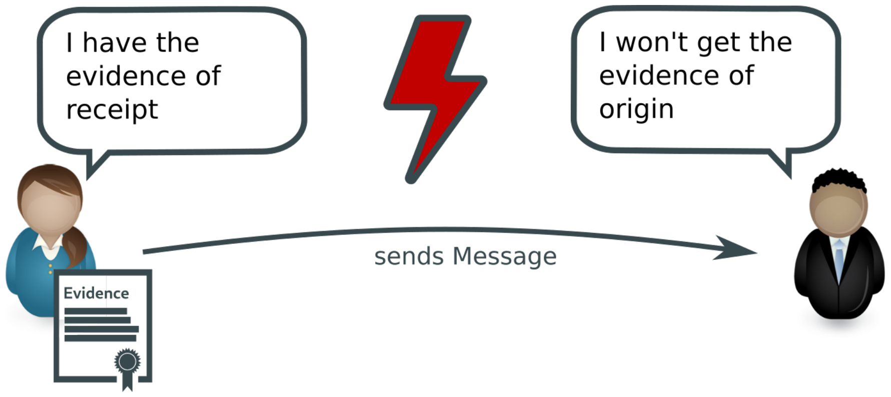

I have the evidence of receipt
I won't get the evidence of origin
sends Message
Evidence

Figure 5.1: Failed non-repudiation for Bob

ecute the protocols in the correct order, hence huge branching may occur. Intuitively, such exchange protocols have great similarity with games, which is why we consider non-repudiation protocols as games between the participants of the protocol.

A good logic to describe games is alternating-time temporal logic (ATL). In ATL we are able to evaluate the strategies of the participants. For an ATL formula $\langle\langle a_1, a_2\rangle\rangle \Box goal$, it is sufficient that agent a_1 and a_2 can enforce computations on which globally *goal* holds. Therefore, we allow systems with computations on which $\Box goal$ does not hold. We simply require that agents a_1 and a_2 can influence the system such that $\Box goal$ holds. In the following, we will analyze non-repudiation properties specified in ATL.

5.2 ATL for Non-Repudiation

Let us introduce the following simplifications. We call the state where non-repudiation of origin or receive is reached NRO or NRR. While analyzing a concrete protocol, one may replace these placeholders with actual evidences of receive or send such as a signature. In order to explain non-repudiation and fair exchange however, using NRO and NRR as shorthands is sufficient. As we mentioned before, we have to consider the behavior of channels. Therefore, channels will be modeled as participants in the protocol. An unreliable channel can therefore decide if he wants to deliver a message or not. As participants respectively agents, we consider a, b and *cha*. For a message a send to b, following [KR03], the main formula enforcing non-repudiation is

$$\textit{non-rep} \coloneqq \neg\langle\langle a, cha\rangle\rangle \Diamond(NRR \wedge \neg\langle\langle b\rangle\rangle \Diamond NRO).$$

Inside the first negation, it describes a situation which should not happen if non-repudiation has to hold. Participant a, together with the communication channels should not reach a state in which

- NRR holds, hence a has the non-repudiation evidence of receipt and

- participant b alone cannot enforce a computation in which he eventually also gets his evidence of origin.

A protocol on which this specification holds can be considered as non-repudiation safe. However, it is not foolproof as Alice (a) may make mistakes, such that it could be to her own disadvantage. According to [KR02], a formula where Alice is not allowed to make mistakes to her own disadvantage is

$$\neg\langle\langle a_h, b, cha\rangle\rangle \Diamond(NRO \wedge \neg\langle\langle\emptyset\rangle\rangle \Diamond NRR).$$

As before, this formula describes a situation that should not happen. All agents working together should not be able to reach a state in which

- Bob has the NRO evidence and
- there is now way such that NRR is reached.

For that we have to assume that Alice is honest, denoted with a_h, meaning that Alice is following the protocol. If Alice would not follow the protocol, we cannot make guarantees at all.

Giving Alice the freedom to make mistakes to her own disadvantage has advantages and drawbacks. On the one hand, at the end of designing a protocol, the protocol should be foolproof. Assumed it is implemented correctly, no mistakes at all should be possible. On the other hand it may occur that, depending on further constraints, no protocol exists fulfilling the stricter non-repudiation. Using the weaker non-repudiation, we therefore can verify and synthesize protocols which specifications otherwise would be unrealizable.

5.3 Future Work on Synthesizing Smart Contracts

For future work to achieve the goal of synthesizing smart contracts respectively non-repudiation protocols we give the following impulses.

For most protocols, cryptographic primitives, such as symmetric or asymmetric key encryption respectively decryption, cryptographic hash functions and signature functions are necessary. For synthesis one has to accurately reflect which of these primitives are necessary for the kinds of protocols one would like to create. Additionally, it has to be defined which constellations of such primitives and received messages constitute the non-repudiation evidences of origin and receive. Also one has to think of the abilities of the trusted third party and how to specify these and how the TTP is allowed to interact with the participants. This also raises the question whether to consider optimistic protocols, where the TTP only acts in case of a conflict. Some of this considerations where already made by the authors of [CR12].

Quite interesting is also the question how we can reconstruct a protocol from the output of a synthesis algorithm. ATL synthesis will output an alternating transition system or concurrent game tree as explained in the previous chapters. Protocols, however, are often noted in text / graphic based form. Therefore, the output system somehow needs to be transformed to this text based representation of a protocol.

In addition one could analyze other properties of contract-signing protocols such as abuse-freeness and balance. Suppose a contract between Alice and Bob is concluding. At some point either the successful completion of the contract or the abort of the contract depends on a decision from Alice or Bob. Abuse-freeness describes the guarantee that it is not possible for neither Alice nor Bob to prove to a third party (Charlie) that the completion or abort depends on their decision. Normally, one wants to prevent such situations as it could give, for example, Alice, the power to force Charlie and Bob to raise his offer. Such situations are comparable to auctions and normally not wanted for regular contract signing. In [KR02], the authors showed that ATL is a good fit for the verification of these properties as well. Therefore, it is promising to analyze these properties as well and how they can be transferred to the synthesis problem.

In general, the fact that ATL is very well suited for abuse-freeness and non-repudiation, shows that ATL is powerful enough to cover a wide variety of properties and is a good fit for properties of smart contracts and exchange protocols in general. It is likely that further properties which are useful for smart contracts are also expressible in ATL.

Chapter 6

Conclusion

We compare both approaches and study which benefits come from the approaches in comparison to existing work. Moreover, we look at further work and improvements to our approaches.

6.1 Reduction From Concurrent Game Structures

In Chapter 3, we presented a reduction from ATL* synthesis to (Hyper-)LTL synthesis. We presented a technique to transform any ATL* formula into a (Hyper-)LTL specification, such that systems modeling these (Hyper-)LTL specifications can be transformed to concurrent game structure which also model the original ATL* specification.

We created this reduction by first transforming concurrent game structures into explicit models and then into computation trees. We used additional LTL formulas and one HyperLTL formula to make sure that the computation tree represents such a concurrent game structure. As a first step we identified all basic formulas of an ATL* specification. We then transformed concurrent game structures into basic concurrent game structures containing propositions for basic formulas. Secondly, we unraveled these basic models to tree models. We then reasoned why it is necessary to perform a boolean widening on this tree, such that the necessary amount of witnesses can be encoded into the tree. As a next step we created explicit models by encoding witnesses and counter witnesses into the tree. A set of formulas guarantees that these explicit models are well-formed. In the second part of this chapter, we transformed the explicit models into computation trees and introduced LTL and HyperLTL formulas enforcing the correctness of the encoding and the well-formedness of the computation trees. For the encoding, we first introduced propositions which map directions of the computation tree to directions of the concurrent game tree, hence input sets to move vectors. We then introduced propositions to encode the relation between moves, move vectors and agents, while a new LTL formula ensures that these relations comply with the definitions of concurrent game structures. Next, we introduced a HyperLTL formula which ensures that the boolean widening we applied to the concurrent game tree is transferable to computation trees. As a next step we created formulas transferring the well-formedness of explicit models to computation trees. These formula implicitly encode the witnesses. As a last step, we create formulas assembling the encoded witnesses and requiring a basic formula to hold on these paths, accordingly to the semantics definition of ATL. We then assemble all formulas, resulting in one complete (Hyper-)LTL specification.

The specification can be entered into any LTL synthesis tool, for the HyperLTL part, we use a tool for bounded HyperLTL synthesis, such as BoSyHyper [Fin+18]. The output represents a concurrent game structure, if necessary it can be reconstructed to a concurrent game structure according to our reduction.

6.2 Reduction From Alternating Transition Systems

In Chapter 4, we presented a second reduction from ATL* synthesis to (Hyper-) LTL synthesis. We presented a technique to transform any ATL* formula into a (Hyper-) LTL specification, such that systems modeling these (Hyper-)LTL specifications can be transformed to trees of alternating transition systems which also model the original ATL* specification.

We created this reduction by first encoding trees of alternating transition systems into regular computation trees. Additional LTL formulas were necessary to ensure the correctness of the encoding. As a first step, we mapped sets of input variables to directions. Secondly, we encoded the behavior function of ATS trees into the computation tree. An additional LTL formula guarantees that the encoding is correct. Next, we encode the witnesses. For that we first need to apply a boolean widening to the tree, to be able to encode the necessary amount of witnesses in each node. An additional HyperLTL formula, ensuring that the widening cannot damage the behavior of the agents, is added here. As a next step, we encoded the witnesses of satisfaction for each basic formula. Based on the ATL semantics definition, defined over ATS trees, we consider strategies and plays as witnesses. We, therefore, introduce propositions for strategies and plays, together with LTL formulas, that ensure that the encoding is correct. As a last step, we introduce formulas assembling the encoded plays to witnesses and requiring a basic formula to hold on these paths, according to the semantics definition of ATL. We then assemble all formulas, resulting in one complete (Hyper-)LTL specification.

Similar to the first approach, this specification can be entered into any LTL synthesis tool. For the HyperLTL part, we use a tool for bounded HyperLTL synthesis, such as BoSyHyper [Fin+18]. The output represents trees of alternating transition systems, and if necessary can be reconstructed to such ATS trees according to our reduction.

6.3 Comparing Both Approaches

We gave two approaches for the reduction, the first one uses concurrent game structures as underlying structures, while the second uses alternating transition systems and its trees. Both approaches were successful and both can be applied in praxis. As the underlying structures were different, the results of both approaches differ as well. Although both approaches result in a set of traces / computation tree, different information are encoded into these trees. Results from the first approach can be transformed back to concurrent game structures, results from the second approach back into alternating transition systems. A main difference is the complexity of the approaches. While the approach over concurrent game structures gives in general an exponential blow-up to the synthesis problem, the approach over trees of alternating transition systems gives a double exponential blow-up to the ATL* synthesis. This is due to the more complex transition function of alternating transition systems. Even though it might be possible to create an encoding for alternating transition systems without this additional exponential blow-up, the straight forward approachesoach we chose has to iterate over the elements of the target set of the transition function, that is 2^{2^I} elements, generating the blow-up.

6.4 Future Work and Optimizations

As mentioned before, it is likely that optimizations of our approaches are possible and may even result in complexity improvements. In [BSK17], the authors showed that for the CTL* synthesis via LTL synthesis case, it is possible to remove the exponential blow-up. It is most likely that a similar approach is possible in our case as well. One also might find a similar technique to remove the HyperLTL formula, such that we will not need HyperLTL at all.

Secondly, implementing the approaches would be the continuation of this work. Since the second approach has better complexity results, we recommend implementing this approach. However, it would be interesting to implement both approaches and analyze how they interact with optimizations of existing LTL synthesis tools.

Besides optimizing and implementing the approach, its application in the context of smart contracts is quite interesting as we outlined in ➔Subsection 5.3. One might consider additional properties such as abuse-freeness and balance, as well as combining them with classical properties such as authenticity and data integrity. Furthermore one has to consider the specification of sender, receiver and trusted third parties, as well as communication channels. Moreover, the output of ATL synthesis, typically concurrent game structures or alternating transition systems, has to be analyzed and transformed to protocol instructions in the typical text representation.

➔Sec. 5.3, Page 59

Bibliography

[AHK98] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. "Alternating-Time Temporal Logic". In: *Technical Reports (CIS)* (Jan. 1, 1998). URL: https://repository.upenn.edu/cis_reports/102.

[AHK02] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. "Alternating-Time Temporal Logic". In: *Journal of the ACM* 49.5 (Sept. 2002), pp. 672–713. ISSN: 0004-5411. DOI: 10.1145/585265.585270. URL: http://doi.acm.org/10.1145/585265.585270 (visited on 05/19/2019).

[ATR19] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper B. Rasmussen. "The {KNOB} is Broken: Exploiting Low Entropy in the Encryption Key Negotiation Of Bluetooth BR/EDR". In: 28th {USENIX} Security Symposium ({USENIX} Security 19). 2019, pp. 1047–1061. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/antonioli (visited on 08/20/2019).

[BSK17] Roderick Bloem, Sven Schewe, and Ayrat Khalimov. "CTL* synthesis via LTL synthesis". In: *Electronic Proceedings in Theoretical Computer Science* 260 (Nov. 28, 2017), pp. 4–22. ISSN: 2075-2180. DOI: 10.4204/EPTCS.260.4. arXiv: 1711.10636. URL: http://arxiv.org/abs/1711.10636 (visited on 03/26/2019).

[Blu83] Manuel Blum. "Coin Flipping by Telephone a Protocol for Solving Impossible Problems". In: *SIGACT News* 15.1 (Jan. 1983), pp. 23–27. ISSN: 0163-5700. DOI: 10.1145/1008908.1008911. URL: http://doi.acm.org/10.1145/1008908.1008911 (visited on 08/20/2019).

[CA07] Krishnendu Chatterjee and Thomas A. Henzinger. "Assume-Guarantee Synthesis". In: TACAS 2007. Vol. 4424. Mar. 24, 2007, pp. 261–275. DOI: 10.1007/978-3-540-71209-1_21.

[CR12] Krishnendu Chatterjee and Vishwanath Raman. "Synthesizing Protocols for Digital Contract Signing". In: *Verification, Model Checking, and Abstract Interpretation.* Ed. by Viktor Kuncak and Andrey Rybalchenko. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 152–168. ISBN: 978-3-642-27940-9.

[Cla+14] Michael R. Clarkson et al. "Temporal Logics for Hyperproperties". In: *Principles of Security and Trust*. Ed. by Martín Abadi and Steve Kremer. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 265–284. ISBN: 978-3-642-54792-8.

[Coe+19] Norine Coenen et al. "The Hierarchy of Hyperlogics". In: LICS. Vancouver, 2019.

[EH86] E. Allen Emerson and Joseph Y. Halpern. ""Sometimes" and "not never" revisited: on branching versus linear time temporal logic". In: *Journal of the ACM (JACM)* 33.1 (Jan. 2, 1986), pp. 151–178. ISSN: 0004-5411. DOI: `10.1145/4904.4999`. URL: `http://dl.acm.org/citation.cfm?id=4904.4999` (visited on 08/13/2019).

[EY80] Shimon Even and Yacov Yacobi. *Relations among public key signature systems*. Computer Science Department, Technion, 1980.

[FFT17] Peter Faymonville, Bernd Finkbeiner, and Leander Tentrup. "BoSy: An Experimentation Framework for Bounded Synthesis". In: *Computer Aided Verification*. Ed. by Rupak Majumdar and Viktor Kunčak. Lecture Notes in Computer Science. Springer International Publishing, 2017, pp. 325–332. ISBN: 978-3-319-63390-9.

[Fin+18] Bernd Finkbeiner et al. "Synthesizing Reactive Systems from Hyperproperties". In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Lecture Notes in Computer Science. Springer International Publishing, 2018, pp. 289–306. ISBN: 978-3-319-96145-3.

[GJM99] Juan A. Garay, Markus Jakobsson, and Philip MacKenzie. "Abuse-Free Optimistic Contract Signing". In: *Advances in Cryptology — CRYPTO' 99*. Ed. by Michael Wiener. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1999, pp. 449–466. ISBN: 978-3-540-48405-9.

[KR02] S. Kremer and J.- Raskin. "Game analysis of abuse-free contract signing". In: *Proceedings 15th IEEE Computer Security Foundations Workshop. CSFW-15*. Proceedings 15th IEEE Computer Security Foundations Workshop. CSFW-15. June 2002, pp. 206–220. DOI: `10.1109/CSFW.2002.1021817`.

[KMZ02] Steve Kremer, Olivier Markowitch, and Jianying Zhou. "An intensive survey of fair non-repudiation protocols". In: *Computer communications* 25.17 (2002), pp. 1606–1621.

[KR03] Steve Kremer and Jean-François Raskin. "A game-based verification of non-repudiation and fair exchange protocols". In: *Journal of Computer Security* 11.3 (Sept. 2003), p. 399. ISSN: 0926227X. DOI: `10.3233/JCS-2003-11307`. URL: `http://search.ebscohost.com/login.aspx?direct=true&db=buh&AN=9972867&lang=de&site=ehost-live` (visited on 03/18/2019).

[Low95] Gavin Lowe. “An attack on the Needham-Schroeder public-key authentication protocol”. In: *Information Processing Letters* 56.3 (Nov. 10, 1995), pp. 131–133. ISSN: 0020-0190. DOI: 10.1016/0020-0190(95)00144-2. URL: http://www.sciencedirect.com/science/article/pii/0020019095001442 (visited on 08/17/2019).

[PVG03] Henning Pagnia, Holger Vogt, and Felix C. Gärtner. “Fair Exchange”. In: *The Computer Journal* 46.1 (Jan. 1, 2003), pp. 55–75. ISSN: 0010-4620. DOI: 10.1093/comjnl/46.1.55. URL: https://academic.oup.com/comjnl/article/46/1/55/415444 (visited on 08/20/2019).

[Pnu77] A. Pnueli. “The temporal logic of programs”. In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 18th Annual Symposium on Foundations of Computer Science (sfcs 1977). Oct. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[PR89] 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*. POPL ’89. event-place: Austin, Texas, USA. New York, NY, USA: ACM, 1989, pp. 179–190. ISBN: 978-0-89791-294-5. DOI: 10.1145/75277.75293. URL: http://doi.acm.org/10.1145/75277.75293 (visited on 08/20/2019).

[Sch08] Sven Schewe. “ATL* Satisfiability Is 2EXPTIME-Complete”. In: *Automata, Languages and Programming*. Ed. by Luca Aceto et al. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 373–385. ISBN: 978-3-540-70583-3.

[SF07] Sven Schewe and Bernd Finkbeiner. “Distributed Synthesis for Alternating-Time Logics”. In: *Automated Technology for Verification and Analysis*. Ed. by Kedar S. Namjoshi et al. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 268–283. ISBN: 978-3-540-75596-8.