Radboud University Nijmegen

Faculty of Science

Extended Liveness Capabilities For Synthesis In CIF

A Rijkswaterstaat case study

Thesis MSc Software Science

Author:
bcs. T.G.M. Beijloos
(Terence)

Supervisor:
dr. ir. D. Hendriks (Dennis)

Second reader:
prof. dr. F.W. Vaandrager
(Frits)

December 2025

Contents

1 Abstract

Synthesis based engineering (SBE) is an approach in which supervisor synthesis is utilized to automatically generate a correct-by-construction supervisor for a cyber-physical system. The Eclipse Supervisory Control Engineering Toolkit (ESCET™) project develops a state-of-the-art toolkit supporting the entire SBE workflow mainly through the CIF language and tools. Rijkswaterstaat (RWS) utilizes SBE and CIF to develop supervisors for complex systems such as waterway locks. However, RWS encountered a challenge during experimentation with requirement state invariants. After the addition of such an invariant, the synthesized supervisor did not allow a waterway lock gate to be opened or closed from certain reachable states in the controlled system. The standard liveness constructs of CIF proved insufficient to guarantee the ability to open and close a lock gate from all reachable states. The absence of such guarantees motivated the work of this thesis to extend CIF with additional liveness capabilities. We researched the state of the art in supervisory control theory regarding liveness requirements in a literature study and formulated two possible approaches to guarantee the reachability of multiple desired states: an algorithmic approach adapting the synthesis algorithm to incorporate the desired liveness requirements, and a novel model-based approach utilizing existing CIF constructs and classical Ramadge and Wonham (RW) supervisory control theory to express the desired liveness requirements. Due to the novelty of the model-based approach, we proved its correctness using the RW-framework in the context of CIF. Lastly, we compared both approaches experimentally on a set of 15 benchmark models which either directly model real cyber-physical systems or are inspired by them. The experiments clearly show the superiority of the algorithmic approach over the model-based approach as it is on average 7.6 times faster and uses 1.2 times less memory. Therefore, the algorithmic approach has been adopted in CIF v10.0 to support liveness requirements semantically expressing CTL formula $AG\ EF\ \phi$, where ϕ is a predicate capturing desired states. This feature solves the challenge faced by RWS and strengthens the correct-by-construction notion of synthesis by providing more precise conditions under which a supervisor is considered correct.

2 Acknowledgements

First and foremost, I would like to express great gratitude to my supervisor, Dennis Hendriks, for his continuous support, guidance, and detailed feedback throughout the entire process of this thesis. His experience and commitment greatly contributed to the quality of this work and inspired me as a person. Secondly, I would like to thank TNO and colleagues for their support and for providing a pleasant and welcoming working environment. I would like to specifically thank Wytse Oortwijn for his supervision during the periods when Dennis was unavailable. Furthermore, I would like to thank Rijkswaterstaat and especially Piotr Klimczak for providing the case study and for their collaboration. I also want to thank Frits Vaandrager for agreeing to be the second reader of this thesis. Additionally, I would like to thank the Radboud University for providing the Software Science Master's program and for the education I received during my studies. Finally, I would like to thank my girlfriend, family, and friends for their support throughout my studies.

3 Introduction

The increasing complexity and reliability-requirements of modern cyber-physical systems (CPSs) present significant challenges for ensuring their correctness and safety. Such systems are crucial in various fields, including semiconductor manufacturing (e.g., wafer handlers) [1], medical imaging (e.g., MRI scanners) [2], and infrastructure (e.g., waterway locks) [3]. Cyber-physical systems are controlled by supervisory controllers, or supervisors for short. Such supervisors ensure the safe operation of CPSs by managing their behavior in response to various inputs and conditions.

Traditional development methods for supervisors, such as manual programming and testing, face significant challenges to reach the desired levels of reliability. Consider for example the Algera bridge in Krimpen a/d IJssel (NL), in this case a supervisor needs to consider $8.4 \cdot 10^{25}$ possible states to ensure correct operation [4]. Reasoning about such an enormous state-space is infeasible for human engineers, making manual programming error-prone. Modern techniques, like model based testing [5], offer improvements, but they still contain fundamental limitations. Such techniques can only show the presence of bugs, not their absence. The presence of bugs in cyber-physical systems can have severe consequences, both on an economic and societal level. Consider for example the malfunctioning of a wafer handler in a semiconductor manufacturing plant, leading to production delays and financial losses for the company. Similarly, a failure in an MRI scanner can put the patient in danger. In infrastructure systems, such as waterway locks, a malfunctioning supervisor can disrupt transportation and logistics, affecting the broader economy and could lead to significant safety risks. Therefore, ensuring the correctness of these systems is crucial for both companies and society.

Synthesis based engineering (SBE) offers a promising solution to reach the desired level of reliability and additionally speed up the development process of supervisors. Instead of manually programming a supervisor, SBE enables engineers to specify the system's behavior and the desired requirements in a formal manner. In doing so, SBE moves the focus from **how** a system should operate, to **what** the system should achieve. The first step in SBE is modeling a specification, which is composed of two parts, a plant model and requirements. A plant model represents all possible behavior of a system, whereas the requirements specify restrictions. From the specification, a supervisor can be synthesized and composed with the plant to form the controlled system, in which the plant's behavior is restricted to ensure the requirements are obeyed. The controlled system is guaranteed correct-by-construction, satisfying four properties: safety, controllability, non-blockingness, and maximal permissiveness [6]. *Safety* means the supervisor will block the system from reaching an unsafe state by restricting transitions or initial states, where the safety of a state is based on the specification. *Controllability* ensures the supervisor will never disable uncontrollable events, when they are enabled in the specification. Uncontrollable events, as the name suggests, are events that cannot be directly disabled by the supervisor. For example, a supervisor can control the opening and closing of a waterway lock gate, but it cannot prevent a sensor from detecting whether a gate is open or closed. *Non-blockingness* guarantees that the system is always able to reach an "interesting" state, formally known as a marked state. *Maximal permissiveness*, also known as *minimal restrictiveness*, ensures the supervisor will only restrict the system when it is strictly necessary to satisfy the other three properties. Once the specification is complete and the supervisor is synthesized, their combination, called the controlled system, can be validated for example through simulation. When the controlled system behaves as desired, the supervisor can be implemented, possibly using code generation, and deployed on the target CPS.

The Eclipse Supervisory Control Engineering Toolkit (ESCET™) project [7, 8, 9] develops a state-of-the-art toolkit supporting the entire SBE workflow, from modeling specifications to synthesizing supervisors, validating controlled systems through simulation and more. While the ESCET toolkit offers a rich set of features, in this thesis we will focus on its SBE capabilities, and specifically the modeling language CIF [10].

The ESCET toolkit is used by Rijkswaterstaat (RWS), the Dutch governmental organization responsible for the design, construction, management, and maintenance of the main infrastructure facilities in the Netherlands, to develop supervisor models for, among other CPSs, waterway locks [3]. RWS uses the synthesized supervisor models to analyze requirement sets. While experimenting with state requirement invariants, during the validation of a requirement set using a synthesized supervisor model, RWS identified reachable states from where a waterway lock gate could no longer be opened or closed. The standard definition of non-blockingness was insufficient to prevent this issue, as both the open- and close state are desired to be marked. Consequently, a marked state was always reachable and thus non-blockingness was satisfied. This issue sparked interest in extending the liveness specification capabilities of CIF, where liveness requirements ensure that something "good" can eventually happen. Being able to specify extended liveness requirements would enable engineers at RWS to incorporate such requirements in their specifications, preventing similar issues in the future. Furthermore, such requirements reduce validation efforts as they provide more precise conditions under which the supervisor is considered correct.

This specific issue highlights a gap in the ability of CIF to handle more sophisticated liveness specifications. To address this gap, we aim to answer the following research questions: *How can CIF be extended to support liveness requirements that require multiple states to be reachable in the controlled system, while preserving safety, controllability, non-blockingness, and maximal permissiveness as they are currently defined?*

Before answering this research question we first provide general background information on supervisors and synthesis in Sections 4.1 and 4.2 respectively. Subsequently, we introduce supervisor synthesis in CIF specifically (Section 4.3). Afterwards, in Section 5, we further elaborate on the challenge encountered by RWS that motivated this research. Then in Section 6, we introduce supervisor sysnthesis, as used in CIF, in a formal manner. Additionally, we formally introduce the modal logics Linear Temporal Logic (LTL) [11] and Computation Tree Logic (CTL) [12], which are used to express liveness requirements in this thesis. Having established sufficient background knowledge, we proceed to conduct a literature study in Section 7 to explore existing methods for specifying and synthesizing liveness requirements in supervisory control theory. We specifically investigate solutions for the Ramadge and Wonham (RW) framework [6] of supervisor synthesis, as CIF's synthesis algorithm is based on this framework. Based on the findings of this study, we propose two approaches in Section 8 to extend the liveness specification capabilities of CIF. Both approaches enforce the same desired liveness specification, but one is algorithmic-based while the other is model-based and can be used in the classical RW-synthesis framework. The first approach is known as "coloring", or "multitasking". In this method, states can be "colored", where from all reachable states of the controlled system, at least one state of each color must be reachable. We slightly deviate from the original multitasking approach [13] by enabling the "coloring" of predicates. Meaning a state where such predicate holds must always be reachable. We denote this method as the "reachability requirement annotation", or "reachability annotation" for short. The second approach captures the liveness specifications as requirement automata, fully utilizing classical concepts of RW-synthesis. We will denote this method as the "reachability requirement automaton", or "reachability automaton" for short. To the best of our knowledge, this is the first time such an approach has been proposed in the context of RW-synthesis. Due to the novelty of this method, we will provide a formal proof of its correctness in Section 9. In this proof, we establish that the reachability requirement automaton preserves safety and utilizes controllability and non-blockingness to minimally restrict the controlled system such that it enforces the reachability of a desired state. By instantiating the requirement automaton multiple times, we can enforce the reachability of multiple desired states.

Finally, in Section 10, we compare both approaches using models which either directly model real cyber-physical systems or are inspired by them, including the waterway lock case from RWS. During this comparison, we evaluate the performance and state-space of both approaches and compare them with each other and with the original specification, that is, without additional liveness requirements. Based on the experimental results, we discuss the strengths and weaknesses of both approaches in Section 11, and provide concluding remarks and directions for future work in Section 12.

4 Background

In this section, we provide the necessary background on the concepts and formalisms of synthesis based engineering in general and for CIF specifically. Later, in Section 6, we will introduce concepts in a formal manner as required for the literature study (Section 7) and proofs in Section 9.

4.1 Supervisors

A *supervisory controller* (or *supervisor*[1]) is a high-level controller which oversees and directs the behavior of a *cyber-physical system* (CPS). Figure 1 shows the typical high-level structure of a CPS under supervisory control. A cyber-physical system is in its physical part composed of mechanical components, which are controlled by actuators, which are further directed by resource controllers. Such resource controllers receive commands from the supervisor(s) based on the environment or based on inputs propagated through a (graphical) interface from human operators / larger systems. Conversely, mechanical components or the environment provide input to sensors which communicate them through resource controllers to supervisor(s). Supervisors can decide whether to communicate the sensor data via a (graphical) interface to human operators / larger systems or act upon the data by sending commands to resource controllers which cascade down to the mechanical components.

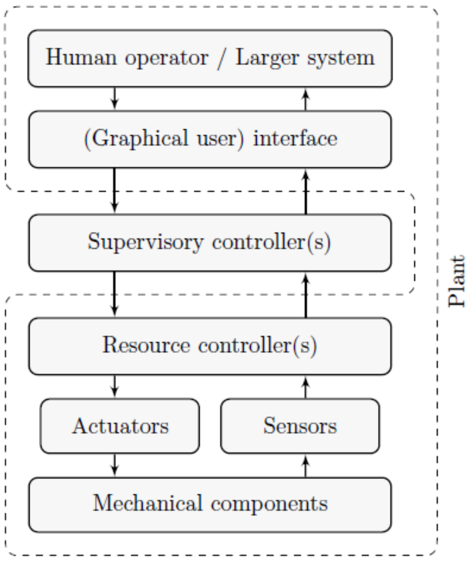

Human operator / Larger system
(Graphical user) interface
Supervisory controller(s)
Resource controller(s)
Actuators
Sensors
Mechanical components
Plant

Figure 1: Structure of a cyber-physical system under supervisory control. This figure is adopted from [9].

The role of a supervisor is formulating and directing output based on input from both mechanical components and from human operators / larger systems. Typically, a supervisor ensures such output is safe, by responding accordingly to actions received from sensors or human operators / larger systems which could lead to unsafe states in the mechanical components.

[1]Technically, in a controller a decision is made which operation to execute given the current state and inputs, whereas a supervisor indicates what decisions are allowed. In this thesis we only consider supervisors.

The environment in which a supervisor operates is called a *plant* or *uncontrolled system*. Figure 1 highlights the components of a plant using a dashed line. The composition of a supervisor with the uncontrolled system is called the *controlled system*.

4.2 Supervisor Synthesis

Supervisor synthesis [6, 14], or *synthesis*, is the automated process of constructing a correct-by-construction supervisor from a formal model of the uncontrolled system and requirements. The formal models of the plant and requirements combined are called the *specification*. Modeling of the specification is typically done using discrete event systems (DES) [15]. A DES is a formal model which represents a system as a finite set of states and transitions between those states, where transitions are triggered by discrete events. Such models reduce complexity of continuous systems by simplifying the system to discrete changes, making formal reasoning easier.

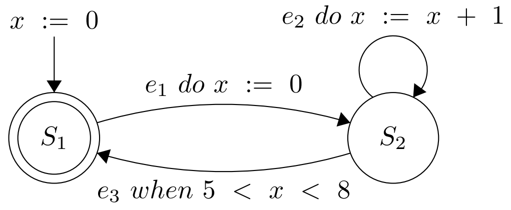


Figure 2: Example of an extended finite automaton.

An extended finite automaton (EFA) is a formalism to describe a DES which extends traditional finite automata with variables, guards, and marked states. An example of an EFA is shown in Figure 2. In this EFA, there are two states S_1 and S_2, with S_1 being the initial state (indicated by the sourceless incoming arrow) and also a marked state (indicated by the double circle). A marked state, in the context of supervisor synthesis, indicates an “interesting” state where all reachable states in the controlled system should be able to reach at least one marked state. Edges between states are labeled with an event name, e_1, e_2, e_3 in this instance, and may include a guard and an action. A guard is formulated as `when <condition>`, where `<condition>` is a predicate which must hold for the edge to be possible (`when 5 < x < 8` in the example EFA). The guards of edges are omitted when they are always true. Actions, indicated by the keyword `do`, are assignments to variables that are performed when the edge is taken. EFAs offer a convenient way to model specifications utilizing both discrete states and variables.

Synthesis algorithms may additionally support *invariants*, which along side EFAs, help to model the specification conveniently. An invariant is a predicate that must always hold. Consider for example an invariant with predicate $x < 10$ for the EFA depicted in Figure 2. This invariant would limit the number of times the edge labeled e_2 can be taken to at most 9 times in a row.

Modern supervisor synthesis takes a specification, modeled using EFAs and invariants, as input and produces a correct-by-construction supervisor EFA as output. The resulting EFA is then composed with the uncontrolled system to form the controlled system. Due to the composition, edges are disabled that would lead to a *safety*, or *non-blockingness* violation of the requirements. A safety violation occurs when an action is taken that leads to violation of requirements. Consider for example again the EFA depicted in Figure 2 with invariant $x < 10$. Taking the edge labeled e_2 when $x = 9$ would lead to a safety violation, as the action $x := x + 1$ would lead to $x = 10$, violating the invariant. A non-blockingness violation occurs when it is no longer possible to reach a marked state from the current state. In the above example EFA, when e_2 is performed 8 times in a row, reaching $x = 8$, no further edges can be taken to reach the marked state S_1, as the guard from e_3 (requiring $5 < x < 8$) is no longer satisfiable, hence violating non-blockingness. *Controllability* is another important property of supervisors. Events of the uncontrolled system are divided into two disjoint sets, namely, *controllable* and *uncontrollable* events. As a supervisor cannot restrict uncontrollable events, it has to restrict earlier controllable edges or make initial states conditional to prevent safety or non-blockingness violations. Uncontrollable events are crucial in correctly describing the possible interaction between the supervisor and the uncontrolled system. For example, sensor interrupts are typically modeled using uncontrollable events, as the supervisor cannot restrict such actions. Lastly, supervisor algorithms often opt for a *maximal permissive*, or otherwise known as *minimally restrictive*, solution. This means that all restrictions imposed by the supervisor are strictly necessary to ensure safety, non-blockingness, and controllability. The four properties of synthesis, safety, non-blockingness, controllability, and maximal permissiveness, combined equip engineers with a powerful tool to automatically generate optimal supervisors from formal specifications.

4.3 Supervisor Synthesis in CIF

CIF, as part of the ESCET toolkit [7], is a language and toolset supporting all steps of the SBE process [10]. While CIF includes a rich set of modeling features, we will focus on the aspects relevant to supervisor synthesis, and more specifically, narrow the considered concepts to those relevant in this thesis. For a more elaborate introduction to CIF we refer the reader to [9].

4.3.1 The CIF Language

In this subsection, we provide a brief overview of the syntax and semantics of CIF.

Automata: The core modeling construct in CIF is the extended finite automaton, syntactically defined to start with the keyword *plant* or *requirement* followed by the name of the automaton. Such an automaton consists of a finite set of locations, edges between locations, and discrete variables that can hold data. Plant automata are used to model possible behavior while requirement automata are used to model constraints on the behavior.

Locations: *location* is the keyword in CIF to define a location within an automaton. It can be annotated as marked or initial using the keywords *marked* and *initial*, respectively. Furthermore, locations can be referenced by automata as predicate which evaluates to true when the automaton is in the referenced location.

Marked: The *marked* keyword indicates that a location is marked. At least one marked location must always be reachable in the controlled system. When a specification is composed of multiple automata (as is generally the case), the specification is considered marked when all its automata are in marked locations. Thus, all automata must include a marked state, else the overall specification can never be marked.

Initial: The *initial* keyword indicates that a location is an initial location. Similarly to marked locations, a specification is considered to be in an initial state when all its automata are in one of their respective initial locations. Therefore, all automata must include an initial state, else the overall specification has no valid initial state.

Transitions: Transitions between locations are defined using the *edge* keyword. The edge keywords must include an event, and may include a target location, guard condition, and variable updates. When no target location is specified, the edge is considered a self-loop. If no guard is specified, the guard is considered to always hold.

Event: An *event* is a discrete occurrence that can trigger transitions between locations. Events can be defined as either controllable or uncontrollable, using the keywords *controllable* and *uncontrollable*, respectively. We will denote an edge controllable if its event is controllable, and uncontrollable otherwise. The guards of controllable edges can be strengthened during synthesis, while the guards of uncontrollable edges must remain unchanged.

When automata share events they *synchronize* on those events. This means that all automata that reference a shared event in one of their edges must take a corresponding edge when that event occurs. Consequently, an edge with a shared event is only enabled when all other automata also have an enabled edge, that is, in all automata that reference the event there must exist an edge with the shared event in its current location and the guard evaluates to true.

Alphabet: The alphabet of an automaton indicates over which events it synchronizes with other automata that also have the same event in their alphabets. The alphabet of an automaton is normally composed of all events referenced by its edges. However, CIF allows for explicitly specifying the alphabet of an automaton using the *alphabet* keyword. When an event is included in the alphabet of an automaton, but not referenced by any of its edges, it means the event is never enabled. This consequently globally disables the event.

Target location: The *goto* keyword is used to specify the target location of an edge.

Actions: *do* can be added to an edge to include variable updates that are executed when the edge is taken. CIF has a local-read/write and global-read principle, meaning variables can only be updated within the automaton they are defined in, but can be read by any automaton in the specification.

Guards: The keyword *when* can be used to specify a condition that must hold for an edge to be possible/enabled.

Discrete variables: Synthesis requires all variables to have a discrete domain, meaning they can only take values from a finite set. This property is required to ensure that the state-space of the system is finite. The types of variables supported for synthesis are booleans, enumerations, and bounded integers. A discrete variable declaration starts with the keyword *disc* followed by the variables's type, name, and (optionally) possible initial values.

Algebraic variables: Algebraic variables are variables whose values are determined by algebraic equations rather than discrete assignments. Such variables are defined similarly to discrete variables, but instead of the keyword *disc*, the keyword *alg* is used and instead of an initial value, an expression is provided.

State/event exclusion invariants: State/event exclusion invariants can be specified in CIF in two forms:

- `invariant <event> needs <predicate>`, this invariant form enforces that the specified event can only occur when the given predicate holds.
- `invariant <predicate> disables <event>`, this invariant form enforces that the specified event cannot occur when the given predicate holds.

State/event exclusion invariants can be used to model a plant or requirements, prefixing an invariant with *plant* and *requirement* respectively.

State invariants: State invariants can be specified in CIF using the syntax: `invariant <predicate>`. State invariants can be used to specify a condition that must always hold. Similar to state/event exclusion invariants, state invariants are prefixed with *plant* or *requirement* to denote to which part of the specification it belongs.

Automata definitions: Automata definitions can be defined using the following syntax: `plant/requirement <name> (parameters*):` This feature allows for parameterized automata, enabling the reuse of automaton definitions with different arguments.

Instantiation: To create instances of plant or requirement automata definitions, the following syntax is used: `<name>:<definition-name>(arguments*)`.

Grouping: CIF supports grouping of plant and requirement automata using the *group* keyword. This allows for more structural modeling of the specification by organizing related concepts together.

The listed concepts combined allow for convenient modeling of a specification. All automata and invariants are either part of the plant or part of the requirements, denoted with *plant* and *requirement*, respectively. Possible behavior of the plant can be modeled using automata and invariants. Restrictions on the plant can be similarly modeled using requirement automata and invariants. Plant state invariants can be used to indicate conditions that per definition always hold in the plant, while requirement state invariants indicate conditions that must always hold in the controlled system. State/event exclusion invariants can be used to indicate the possibility, in the case of a plant, or the desirability, in the case of requirements, of events occurring under certain conditions.

Recall the EFA depicted in Figure 2. The CIF equivalent is shown in Listing 1. For the purpose of this example the EFA is assumed to be a plant and e_1 and e_2 are controllable events (line 2), while e_3 is uncontrollable (line 3). Similarly, for illustrative purposes, we assume the invariant $x < 10$ is a requirement invariant (line 17). In the subsequent subsection, we will elaborate on how this specification is used in CIF's supervisor synthesis.

4.3.2 Supervisor Synthesis Process in CIF

In this subsection, we discuss the supervisor synthesis process in CIF. An overview of this process is shown in Figure 3. In the following paragraphs, we will explain each of the steps depicted in the figure. The explanation of each step includes a description of how the step applies to the running example depicted in Listing 1. Notably, not all preprocessing steps are relevant for this specific example. But, we will further elaborate on important steps, for the scope of this thesis, in the preliminaries (Section 6).

```
1   plant P:
2       controllable e1, e2;
3       uncontrollable e3;
4
5       disc int[0..15] x = 0;
6
7       location S_1:
8           initial;
9           marked;
10          edge e1 do x := 0 goto S_2;
11
12      location S_2:
13          edge e2 do x := x + 1;
14          edge e3 when 5 < x and x < 8 goto S_1;
15  end
16
17  requirement invariant P.x < 10;
```

Listing 1: CIF representation of the EFA from Figure 2.

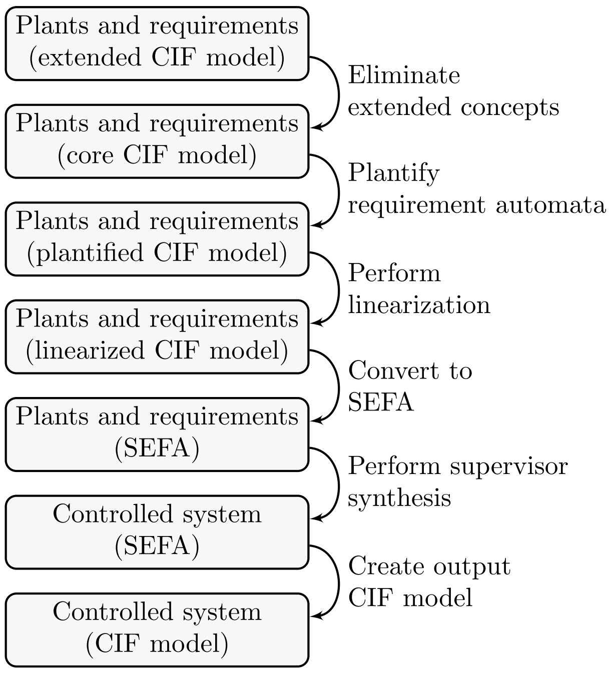

Plants and requirements (extended CIF model)
Eliminate extended concepts
Plants and requirements (core CIF model)
Plantify requirement automata
Plants and requirements (plantified CIF model)
Perform linearization
Plants and requirements (linearized CIF model)
Convert to SEFA
Plants and requirements (SEFA)
Perform supervisor synthesis
Controlled system (SEFA)
Create output CIF model
Controlled system (CIF model)

Figure 3: Internal steps in CIF's symbolic supervisor synthesis tool. This figure is originally from [9].

Plants and requirements: The synthesis process starts with a CIF specification consisting of plant and requirement automata (EFAs) and invariants. CIF is a powerful and feature rich language, which simplifies development but complicates the synthesis algorithm. Consequently, the first two steps are to reduce the model's complexity by translating the specification first into a core CIF model and afterwards into a plantified CIF model.

In this step the model depicted in Listing 1 is the starting point for the synthesis process.

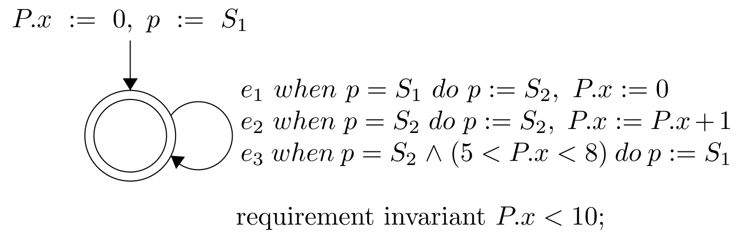
Eliminate extended concepts: This transformation involves inlining constants and algebraic variables, removing groups, combining imports, instantiating parameterized components, and grouping definitions.

As the running example in Listing 1 does not use any of these extended concepts, the model remains unchanged in this step.

Plantify requirement automata: In this step, the concept of requirement automata is eliminated by restructuring them as plants and translating the restrictions they impose to invariants.

In the running example, there are no requirement automata present, so the model remains unchanged in this step as well.

Perform linearization: The next step is to convert the core CIF model into a linearized CIF model. This transformation flattens the hierarchical structure of the automata by creating a single automaton that captures the combined behavior of all automata in the specification. The new structure makes it more suitable to convert to a symbolic extended finite automaton (SEFA) in the next step. Both linearization and the conversion to a SEFA will be explained in more (formal) detail in the preliminaries (Section 6).



$P.x := 0,\ p := S_1$
e_1 *when* $p = S_1$ *do* $p := S_2,\ P.x := 0$
e_2 *when* $p = S_2$ *do* $p := S_2,\ P.x := P.x + 1$
e_3 *when* $p = S_2 \wedge (5 < P.x < 8)$ *do* $p := S_1$
requirement invariant $P.x < 10$;

Figure 4: Linearized CIF model of the running example from Listing 1.

Figure 4 shows the linearized CIF model as EFA of the running example from Listing 1. The variable p is introduced to replace the location information of the original model to maintain its behavior.

Convert to SEFA: The linearized CIF model is then translated into a symbolic extended finite automaton (SEFA) representation. This conversion allows developers to model the specification conveniently using EFAs but take advantage of the efficient symbolic synthesis algorithm that operates on SEFAs. During this transformation, a forbidden state predicate is constructed based on the invariants present in the linearized CIF model.

The resulting SEFA of the running example is similar to the linearized CIF model shown in Figure 4, except that the only remaining location is omitted, and thus the state is fully captured using variables. Additionally, the invariant $P.x < 10$ is translated into a forbidden state predicate $P.x \geq 10$. Furthermore, the domain of variable $P.x$ is $\{0, \ldots, 15\}$ which contributes to the forbidden state predicate as well: $P.x \geq 10 \vee P.x < 0 \vee P.x > 15$.

Perform supervisor synthesis: The symbolic supervisor synthesis algorithm of CIF is applied to the SEFA representation of the specification. During synthesis, the guards of controllable transitions and initial state predicates are strengthened to ensure that the resulting supervisor is safe, controllable, non-blocking, and maximally permissive. This approach of supervisor synthesis is based on [16] and further detailed in [9].

The first step of synthesis applied on the SEFA of the running example results in strengthening the guard of edge e_2 to $(p = S_2) \wedge (P.x < 9) \wedge (P.x < 15)$, due to the forbidden state predicate. This is simplified to $(p = S_2) \wedge (P.x < 9)$. Afterwards, the synthesis algorithm observes that edge e_2 can lead to a blocking state when $P.x >= 7$, as the marked state S_1 can then no longer be reached. To prevent this blocking situation, the guard of edge e_2 is further strengthened to $(p = S_2) \wedge (P.x < 9) \wedge (P.x < 7)$. Which simplifies to $(p = S_2) \wedge (P.x < 7)$. The guard can be even further simplified to $(p = S_2) \wedge (P.x \neq 7)$, as the edge over e_2 increases the value of $P.x$ by 1. This concludes the synthesis step for the running example.

Create output CIF model: Finally, the synthesized SEFA is translated back into a CIF model. This model can then be further analyzed, simulated, or exported to use in other analytic tools.

The resulting CIF model of the running example, including the synthesized supervisor, is shown in Listing 2. The transition using e_1 remains unrestricted, while the guard of transition e_2 is strengthened to $P.x \neq 7$, to prevent reaching a blocking state. The addition of $(p = S_2)$ to the guard is omitted, as this is now captured again through the location of the automaton. e_3 is notably not present in the alphabet of the supervisor, as it is an uncontrollable event.

```
plant automaton P:
  controllable e1, e2;
  uncontrollable e3;
  disc int[0..15] x = 0;
  location S_1:
    initial;
    marked;
    edge e1 do x := 0 goto S_2;
  location S_2:
    edge e2 do x := x + 1;
    edge e3 when 5 < x and x < 8 goto S_1;
end
supervisor automaton sup:
  alphabet P.e1, P.e2;
  location:
    initial;
    marked;
    edge P.e1 when true;
    edge P.e2 when P.x != 7;
end
supervisor invariant P.x < 10;
```

Listing 2: CIF representation of the EFA from Figure 2 along with its synthesized supervisor.

5 Motivating Case Study

In this section, we introduce a challenge encountered at Rijkswaterstaat (RWS) during experimentation with requirements for a waterway lock. This challenge provided the initial motivation for this thesis. We will refer back to it in later sections to illustrate the practical relevance of our results. We first provide background information on the waterway lock specification. Subsequently, we describe the specific challenge faced by RWS engineers.

5.1 Generic Waterway Lock Model

RWS develops CIF models for various types of waterway locks, which are all based on a generic waterway lock model. This generic model describes the possible behavior of the main components of a waterway lock, such as sluice gates, acoustic signaling, navigational lights, and sensors. Additionally, the model includes an extensive set of requirements which ensure the controlled system adheres to safety and operational protocols. Furthermore, the model is designed to be modular, allowing for easy adaptation to different types of waterway locks by adding or removing components as needed. This process is even more simplified through the use of a configuration GUI developed by RWS, which allows engineers to tailor the generic model to specific lock types without needing to modify the underlying CIF code directly.

5.2 Limitations of Non-blockingness

During experimentation with new requirement state invariants for the generic waterway lock model, RWS engineers encountered a case where the controlled system could reach one or more states from which a waterway lock gate could no longer be opened or closed. The challenge started with the introduction of the following requirement state invariant:

```
requirement invariant not
    (GenericLock.UpperHead.DoorName1.Actuator.open and
     GenericLock.LowerHead.DoorName1.Sensor.opened);
```

This code snippet specifies that the actuator responsible for opening the upper sluice gate and the sensor indicating whether the lower sluice gate is opened should never both be active simultaneously. The intention behind this requirement is to prevent a situation where both sluice gates are open at the same time, which could lead to unsafe conditions.

However, after incorporating this requirement into the generic waterway lock model, RWS engineers observed during validation that the controlled system could reach states in which it was no longer possible to open a lock gate. This behavior was later confirmed through model checking using μ-calculus [17] in mCRL2 [18].

In CIF, non-blockingness is defined such that at least one marked state should always be reachable in the controlled system. However, in the context of the waterway lock system, this definition proves insufficient. In this case, both a lock gate being open and a lock gate being closed are desired marked states, so it is possible that one of these states is not always reachable in the controlled system (as synthesis only guarantees that a marked state is reachable, not that they are all reachable). This limitation increases the effort required during validation. Engineers must verify, preferably using model checking, or otherwise through simulation, whether all of their desired states are indeed always reachable in the controlled system. More fundamentally, it weakens the correct-by-construction guarantee of supervisory control theory, as engineers cannot fully rely on the controlled system to capture all desired behaviors. Intuitively, it would be more convenient to express such requirements as liveness requirements, stating that certain desired states (such as an open or closed lock gate) must always remain reachable in the controlled system. When this is not possible, synthesis should result in an empty supervisor, indicating that the liveness requirements are too strict to be satisfied given the plant and other requirements.

This case study at RWS raises the following central research question for this thesis: *How can CIF be extended to support liveness requirements that require multiple states to be reachable in the controlled system, while preserving safety, controllability, non-blockingness, and maximal permissiveness as they are currently defined?*

Based on the gaps and solutions identified in the literature (Section 7), we refine the central question into more specific research questions in Section 7.9.

6 Preliminaries

In this section, we introduce the formal foundations necessary to understand the literature study and the contributions of this thesis. We first, in Section 6.1, establish a notational convention used throughout this thesis. Afterwards, in Section 6.2, we establish the main formalism used in CIF, EFA [19] and SEFA [9]. Subsequently, in the same subsection, we define key concepts for (S)EFAs such as transition relations, languages, runs, parallel composition, and the transformation from EFA to SEFA. Then, in Section 6.3, we define supervisory control theory concepts such as controllability, non-blockingness, safety, and maximal permissiveness. Additionally, we define the construction of a supervisor in CIF based on the SCT framework of Ramadge and Wonham [6]. Finally, in Section 6.4, we provide a brief overview of Linear Temporal Logic (LTL) and Computation Tree Logic (CTL), which are highly relevant concepts in the literature study (Section 7) and subsequent sections.

6.1 Element References

To distinguish elements of different tuples, we denote an element E of tuple T as E_T, where the subscript T indicates the tuple to which E belongs. For elements that already include subscripts, we separate the element's subscript from the tuple reference with a comma. For example, we denote element p_0 of tuple X as $p_{0,X}$. The tuple reference subscript is omitted when it clearly follows from the context.

6.2 Syntax and Semantics of (S)EFAs

6.2.1 Extended Finite Automaton

As described in Section 4.3, CIF primarily uses *Extended Finite Automata* (EFAs) [19] to model system behavior. Intuitively, an EFA extends a classical finite automaton with variables, guards, updates, and in CIF's case also error predicates, allowing both discrete control logic and finite data to be modeled in a single formalism. In the remainder of this thesis, we rely on a precise definition of EFAs in order to formalize concepts such as runs, languages, composition, and supervisory control properties.

Definition 6.1 (Extended Finite Automaton)**.** An **Extended Finite Automaton (EFA)** is formally defined as a 7-tuple $(Q, V, D, \Sigma, E, p_0, p_m)$ where:

- Q: A finite set of locations. The set of locations is not equal to the set of all states, as states also include variable valuations, see the definition of all possible states X below. Each $q \in Q$ can be used in boolean proposition to identify locations.
- V: A finite set of variables.
- D: For each variable $v \in V$, D_v defines its finite domain of possible values.
- X: Denotes the set of possible states, where each state is a combination of a current location and a current value for each variable, $X := Q \times D_1 \times D_2 \times \dots D_n$. A single state $x \in X$ is a tuple which includes a location $q \in Q$ and for each variable v_i a single value from D_{v_i}. The projection $x[Q]$ outputs the corresponding location of x. Similarly, the projection $x[v_i]$ outputs the value of v_i given state x.
- Σ: A finite set of events (the alphabet), partitioned into two disjoint sets: Σ_c for controllable events and Σ_u for uncontrollable events, i.e., $\Sigma := \Sigma_c \cup \Sigma_u$, $\Sigma_c \cap \Sigma_u = \emptyset$.
- E: A finite set of edges (transition relations), partitioned into E_c (edges for events in Σ_c) and E_u (edges for events in Σ_u). Each edge is a 6-tuple (q, g, r, σ, u, q'), where:
  1. q: The source location, $q \in Q$, which is the location from where the transition is taken.
  2. g: Guard predicate over states in X, indicating the states for which the edge is enabled (i.e., a transition is possible).
  3. r: Error predicate over X, specifying states such that the update u would result in values for variables v_i outside their range D_{v_i}.
  4. σ: Event from Σ associated with the edge.
  5. u: Update predicate over $Q \cup V^*$, where $V^* := V \cup V^+$ is composed of old and new variables, V and V^+ respectively. u can reference the current location q as boolean in its predicate but it will not have an influence over the target location q'. Counterpart variables, v^+, denote the post-transition values.
  6. q': The target location, $q' \in Q$, to which is transitioned.

- p_0: Predicate over X representing the initial states, with $X^0 := \{\, x \in X \mid x \models p_0 \,\}$ denoting the corresponding set of initial states.
- p_m: Predicate over X representing the marked states, with $X^m := \{\, x \in X \mid x \models p_m \,\}$ denoting the corresponding set of marked states.

6.2.2 Symbolic Extended Finite Automaton

A *symbolic extended finite automaton* (SEFA), as described by [9], is similar to an EFA but locations are represented by location pointer variables, such that all state is in the variables. By doing so, SEFAs enable a symbolic state-space representation that makes synthesis more efficient and more scalable [9].

Definition 6.2 (Symbolic Extended Finite Automaton)**.** A **Symbolic Extended Finite Automaton (SEFA)** is formally defined as a 6-tuple $(V, D, \Sigma, E, p_0, p_m)$, where:

- V: A finite set of variables.
- D: For each variable $v \in V$, D_v defines its finite domain of possible values.
- X: Denotes the set of all possible states, where each state is a tuple representing a possible variable valuation, $X := D_1 \times D_2 \times \ldots D_n$. A single state $x \in X$ is a tuple which associates each variable v_i with a single value from D_{v_i} where the projection $x[v_i]$ outputs the corresponding value of v_i given state x.
- Σ: A finite set of events (the alphabet), partitioned into two disjoint sets: Σ_c for controllable events and Σ_u for uncontrollable events, i.e., $\Sigma := \Sigma_c \cup \Sigma_u$, $\Sigma_c \cap \Sigma_u = \emptyset$.
- E: A finite set of edges (transition relations), partitioned into E_c (edges for events in Σ_c) and E_u (edges for events in Σ_u). Each edge is a 4-tuple (g, r, σ, u), where:
  1. g: Guard predicate over states in X, indicating the states for which the edge is enabled (i.e., a transition is possible).
  2. r: Error predicate over X, specifying states such that the update u would result in values for variables v_i outside their range D_{v_i}.
  3. σ: Event from Σ associated with the edge.
  4. u: Update predicate over $V^* := V \cup V^+$, where V^* is composed of the old and new state variables, V and V^+ respectively. Counterpart variables, v^+, denote the post-transition values.
- p_0: Predicate over X representing the initial states, with $X^0 := \{\, x \in X \mid x \models p_0 \,\}$ denoting the corresponding set of initial states.
- p_m: Predicate over X representing the marked states, with $X^m := \{\, x \in X \mid x \models p_m \,\}$ denoting the corresponding set of marked states.

6.2.3 Transition Relations

In the next two subsections we define the *transition relations* for both EFAs and SEFAs, as well as the extension of the transition relation to words. Such relations are fundamental to defining languages, runs, and other key concepts for (S)EFAs.

6.2.4 EFA Transition Relation

Consider an EFA, any states $x, x' \in X$ and an event $\sigma \in \Sigma$, we write $x \xrightarrow{\sigma} x'$ if there exists an edge $(q, g, r, \sigma, u, q') \in E$ such that:

1. $x \models q$ (the source location matches).
2. $x \models g$ (the guard condition holds).
3. $x \not\models r$ (the error condition does not hold).
4. $x' \models u(x)$ (the update predicate u applied to x yields x', where $u(x)$ denotes the predicate u with each variable v replaced by its value in x and each v^+ interpreted as the new value in x').
5. $x' \models q'$ (the target location matches).

6.2.5 SEFA Transition Relation

Consider a SEFA, any states $x, x' \in X$ and an event $\sigma \in \Sigma$, we write $x \xrightarrow{\sigma} x'$ if there exists an edge $(g, r, \sigma, u) \in E$ such that:

1. $x \models g$ (the guard condition holds).
2. $x \not\models r$ (the error condition does not hold).
3. $x' \models u(x)$ (the update predicate u applied to x yields x', where $u(x)$ denotes the predicate u with each variable v replaced by its value in x and each v^+ interpreted as the new value in x').

6.2.6 Transition Relation Over Words

Definition 6.3 (Transition relation over words)**.** We extend the transition relation $\rightarrow$ of a (S)EFA to include words $w \in \Sigma^*$ as follows:

1. For the empty word, $w = \epsilon$, we define $x \xrightarrow{\epsilon} x'$ iff $x = x'$.
2. For any word $w = \sigma w'$, where $\sigma \in \Sigma$ and $w' \in \Sigma^*$, we define $x \xrightarrow{w} x'$ iff $\exists x'' \in X$ such that $x \xrightarrow{\sigma} x''$ and $x'' \xrightarrow{w'} x'$.

6.2.7 Language

The *language* of a (S)EFA captures all possible sequences of events (words) that such an automaton can generate starting from an initial state. This concept is fundamental in supervisory control theory, as all four properties (safety, controllability, non-blockingness, and maximally permissiveness) of supervisory control are traditionally defined in terms of languages. Please note that the language defined here includes all possible words, not just the marked words (which are defined in the next subsection).

Definition 6.4 (Language)**.** The **language** of a (S)EFA M, denoted as $L(M)$, is the set of all possible words of M: $L(M) := \{w \in \Sigma^* \mid \exists x_0 \in X^0, x' \in X \ : x_0 \xrightarrow{w} x'\}$

6.2.8 Marked language

The *marked language* of a (S)EFA captures all possible words that start from an initial state and end in a marked state. The marked language is an important notion in the definition of non-blockingness (Definition 6.21), as it is used to identify states from where a marked state can be reached.

Definition 6.5 (Marked language)**.** The **marked language** of a (S)EFA M, denoted as $L_m(M)$, is the set of all marked words M: $L_m(M) := \{\, w \in \Sigma^* \mid \exists x_0 \in X^0,\, x_m \in X^m :\ x_0 \xrightarrow{w} x_m \,\}$.

6.2.9 Runs

The notion of *runs* captures the sequences of states that a (S)EFA traverses when processing words from the provided language. It is useful for reasoning about the states visited given a (S)EFA and a language.

Definition 6.6 (Runs)**.** *Runs* is a function which takes a (S)EFA M and a language $L \subseteq \Sigma^*$ as input, and outputs a set containing a sequence of states for each word in L: $\mathit{Runs}(M, L) := \{x_0, x_1, \ldots, x_k \in X_M \mid w = \sigma_1\sigma_2 \ldots \sigma_k \in L\}$ (where $k \geq 0$), such that:

1. $x_0 \models p_0$,
2. For each $i = 0, 1, \ldots, k-1$, $x_i \xrightarrow{\sigma_{i+1}} x_{i+1}$.

Each sequence $(x_0, x_1, \ldots, x_k)$ in $\mathit{Runs}(M, L)$ is referred to as a *run* of M over its corresponding word w. The i-th state in a run r is obtained by projection, $\mathrm{r}[i] = x_i$.

6.2.10 States

States is a function that extracts all states from a set of runs. This function is useful for reasoning about the collection of states that are visited in a set of runs.

Definition 6.7 (States)**.** *States* is a function which takes a set of state sequences, denoted *runs*, as input and outputs a set containing all states referenced in *runs*: $\mathit{States}(\mathit{runs}) := \{\, \mathit{run}[i] \mid \mathit{run} \in \mathit{runs},\ 0 \leq i < |\mathit{run}| \,\}$.

6.2.11 Total

The *totality property* ensures that from every state in a (S)EFA, there is at least one possible transition. This is a key property required for extending runs indefinitely.

Definition 6.8 (Total)**.** We define a (S)EFA M to be **total** if there exists an enabled outgoing transition for every state. Formally, M is total iff:

$$\forall x \in X, \exists x' \in X, \exists \sigma \in \Sigma : x \xrightarrow{\sigma} x'$$

6.2.12 Closure

The notion of *closure* captures all prefixes of words in a given set of words. This concept is useful in supervisory control theory (Section 6.3), specifically in the definition of non-blockingness (Definition 6.21).

Definition 6.9 (Closure)**.** Consider a set of words W over Σ^*, such that $W \subset \Sigma^*$. The **closure** of W, denoted as $\overline{W}$, is the set of all prefixes of words in W. Formally, $\overline{W} = \{p \in \Sigma^* \mid \exists t \in \Sigma^*, pt \in W\}$, as defined in [6].

6.2.13 Parallel Composition

In CIF, the following definition is used for *parallel composition*, also known as *full synchronous composition*. The definition is given for two EFAs but can be applied repeatedly to create the parallel composition of any number of EFAs. This definition stems from [19], but is slightly adjusted to align with the definition of EFAs in CIF, as described in [9]. Parallel composition enables the modeling of specifications conveniently using multiple EFAs.

Definition 6.10 (Parallel Composition)**.** Consider EFAs M and N represented as tuples, sharing the same variable domain, $V_M = V_N$ and $D_M = D_N$. Consider the parallel composition P, such that $P := M \parallel N$ where:

$$\begin{aligned} Q_P &:= Q_M \times Q_N && \text{(product of location sets)},\\ \Sigma_P &:= \Sigma_M \cup \Sigma_N && \text{(union of event sets)},\\ p_{0,P} &:= p_{0,M} \wedge p_{0,N} && \text{(conjunction of initial states)},\\ p_{m,P} &:= p_{m,M} \wedge p_{m,N} && \text{(conjunction of marked states)}, \end{aligned}$$

The transition relation E_P is defined as follows:

1. **Shared events $\sigma \in \Sigma_M \cap \Sigma_N$:**
   $((q_M, q_N), \sigma, g, r, u, (q'_M, q'_N)) \in E_P$ iff
   - $(q_M, \sigma, g_M, r_M, u_M, q'_M) \in E_M$,
   - $(q_N, \sigma, g_N, r_N, u_N, q'_N) \in E_N$,
   - $g = g_M \wedge g_N$ (combined guard),
   - $r = r_M \vee r_N$ (combined error predicate),
   - the combined update u is given by

   $$u(v) = \begin{cases} u_M(v) & \text{if } u_M(v) = u_N(v)\\ u_M(v) & \text{if } u_N(v) = \Xi \text{ (“don't care”)}\\ u_N(v) & \text{if } u_M(v) = \Xi\\ v & \text{otherwise (conflicting updates)} \end{cases}$$

   Where the “don't care”, Ξ, means the variable is not updated, i.e., implicitly, $v^+ = v$.

2. **Events unique to M, i.e., $\sigma \in \Sigma_M \setminus \Sigma_N$:**
   $((q_M, q_N), \sigma, g, u, (q'_M, q_N)) \in E_P$ iff $(q_M, \sigma, g, u, q'_M) \in E_M$.

3. **Events unique to N, i.e., $\sigma \in \Sigma_N \setminus \Sigma_M$:**
   $((q_M, q_N), \sigma, g, u, (q_M, q'_N)) \in E_P$ iff $(q_N, \sigma, g, u, q'_N) \in E_N$.

6.2.14 Action Consistency

Action consistency is a condition that guarantees deterministic variable update behavior of the composed system by ensuring that shared events do not cause conflicting updates to variables.

Definition 6.11 (Action Consistency)**.** Two EFAs M and N are *action consistent* if for every pair of transitions $(q_M, \sigma, g_M, r_M, u_M, q'_M) \in E_M$ and $(q_N, \sigma, g_N, r_N, u_N, q'_N) \in E_N$ with the same event $\sigma \in \Sigma_M \cap \Sigma_N$, and for all valuations $x \in X$ such that $g_M(x)$ and $g_N(x)$ both evaluate to `true`, we have for each variable $v_i \in V$:

- $u_M(v_i) = u_N(v_i)$, or
- $u_M(v_i) = \Xi$, or
- $u_N(v_i) = \Xi$.

All specifications are from now on assumed to be action consistent, as CIF only allows specifying action-consistent specifications due to the global-read and local-write property (Section 4.3).

6.2.15 EFA to SEFA Transformation

In CIF, synthesis is performed exclusively on SEFAs. A SEFA encodes system behavior as variables, enabling a symbolic state-space representation that makes synthesis more efficient and scalable [9]. However, EFAs offer a more intuitive design process [20] due to their explicit locations. To combine the modeling convenience of EFAs with the efficiency of SEFAs, a transformation from EFA to SEFA is applied before synthesis.

Conceptually, the only structural difference between an EFA and a SEFA is that an EFA contains locations explicitly, while a SEFA represents them as variables. Each location variable, called a location pointer, evaluates to a location of Q, where Q is the set of locations of the original EFA. This symbolic encoding allows all location information to be manipulated through variable updates.

For systems defined as a parallel composition of multiple EFAs, one location pointer variable is introduced for each component, enabling design of the specification using multiple EFAs.

The first step in transforming an EFA into a SEFA is called linearization. Linearization is a transformation from an (parallel composed) EFA $\mathcal{E}$ to an EFA $\mathcal{E}'$, where $\mathcal{E}'$ contains only one location and all location information of $\mathcal{E}$ is captured through variables.

Definition 6.12 (Linearization)**.** Consider EFA $\mathcal{E} := \mathcal{E}_1 \| \mathcal{E}_2 \| \ldots \| \mathcal{E}_n$ and the reference to a parallel component of $\mathcal{E}$, $\mathcal{E}_i$ $(1 \leq i \leq n)$. Then, **linearization** of EFA $\mathcal{E}$ to EFA $\mathcal{E}'$ is defined as follows:

- $Q_{\mathcal{E}'} := \{q\}$, a single location after transformation.
- $V_{\mathcal{E}'} := V_{\mathcal{E}} \cup \{v_1, v_2, \ldots, v_n\}$, a location pointer variable v_i is introduced for each component $\mathcal{E}_i$.
- $D_{\mathcal{E}'}$: For each variable $v \in V_{\mathcal{E}'}$, $D_{\mathcal{E}',v}$ defines its domain. For a location pointer variable v_i, its domain is equal to the set of locations of its original EFA: $D_{\mathcal{E}',v_i} = Q_{\mathcal{E}_i}$.
- $X_{\mathcal{E}'} := X_{\mathcal{E}} \times D_{\mathcal{E}',v_1} \times \ldots \times D_{\mathcal{E}',v_n}$, the possible variable valuations of each location pointer variable are combined with the set of possible states of $\mathcal{E}$. Note that $X_{\mathcal{E}'}$ does not include $Q_{\mathcal{E}'}$, as only a single location remains after linearization and therefore the location set becomes redundant in describing the current state.
- $\Sigma_{\mathcal{E}'} := \Sigma_{\mathcal{E}}$. The alphabet is maintained.
- $E_{\mathcal{E}'} := \{(q, g, r, \sigma, u, q) \mid ((q_1, q_2, \ldots, q_n), g_{\mathcal{E}}, r_{\mathcal{E}}, \sigma, u_{\mathcal{E}}, (q_1', q_2', \ldots, q_n')) \in E_{\mathcal{E}}\}$. Then the edge transformation is defined as follows:
  1. All edges transition from and to the only location remaining.
  2. $g := g_{\mathcal{E}} \wedge \bigwedge_{i=1}^{n} v_i = q_i$. The guard now includes location pointer variables.

3. $r := r_{\mathcal{E}}$. Location pointer variables do not contribute to error predicates as, by definition, they may only be updated as described in u. This update never assigns a value outside a location pointer's range as, by definition, $q_i \in Q_{v_i}$.
4. $u := u_{\mathcal{E}} \wedge \bigwedge_{i=1}^{n} v_i^+ = q_i'$, the location pointer variable is updated to represent the original transition.

- $p_{0,\mathcal{E}'}$ becomes the predicate over $X_{\mathcal{E}'}$ such that $p_{0,\mathcal{E}'} \iff p_{0,\mathcal{E}}$.
- $p_{m,\mathcal{E}'}$ becomes the predicate over $X_{\mathcal{E}'}$ such that $p_{m,\mathcal{E}'} \iff p_{m,\mathcal{E}}$.

Linearization maintains the language and marked language of the transformed EFA, as the transformation is purely structural.

We define the transformation from a linearized EFA to a SEFA as follows:

Definition 6.13 (EFA to SEFA transformation)**.** Consider a function *ToSEFA* which takes an EFA E as input and outputs a SEFA $\mathcal{S}$. Consider the linearized EFA of E, given Definition 6.12, $\mathcal{E}$. Then *ToSEFA*$(E) = \mathcal{S}$ is defined as follows: $\mathcal{E} = \mathcal{S}$, where $(V_{\mathcal{E}}, D_{\mathcal{E}}, \Sigma_{\mathcal{E}}, E_{\mathcal{E}}, p_{0,\mathcal{E}}, p_{m,\mathcal{E}}) = (V_{\mathcal{S}}, D_{\mathcal{S}}, \Sigma_{\mathcal{S}}, E_{\mathcal{S}}, p_{0,\mathcal{S}}, p_{m,\mathcal{S}})$.

The only difference between the linearized EFA $\mathcal{E}$ and result $\mathcal{S}$, is the exclusion of location set $Q_{\mathcal{E}}$. This set can be omitted without any influence on the (marked) language.

6.2.16 Concatenation

Concatenation is an operation that combines two words into a single word by appending one after the other. This operation is useful when reasoning about combined sequences of events in languages.

Definition 6.14 (Concatenation)**.** Consider an alphabet Σ, the concatenation operation $(\cdot)$ on words over Σ is defined as follows. Consider sequences of events u and v, such that $(u_1, u_2, \ldots, u_n)$, $(v_1, v_2, \ldots, v_m) \in \Sigma^*$. Then $u \cdot v$ is defined as $u_1, u_2, \ldots, u_n, v_1, v_2, \ldots,$ v_m. Note that by definition $\epsilon \cdot v = v$ and $u \cdot \epsilon = u$. We write uv as shorthand for $u \cdot v$.

6.2.17 Projection

Projection is an operation that removes certain symbols from words in a language, effectively filtering the language to a smaller alphabet. This operation is useful when reasoning about languages over different alphabets.

Definition 6.15 (Projection)**.** Consider alphabets Σ and Σ' such that Σ' is a strict subset of Σ, i.e., $\Sigma' \subset \Sigma$. Then *Proj* is the projection function which takes a language L over Σ as input, $L \subseteq \Sigma^*$, and outputs the language L' over Σ', $L' \subseteq \Sigma'^*$, such that all symbols not in Σ' are replaced by the empty string ϵ for all words in L. We refer to the output alphabet of *Proj* using subscript, $Proj_{\Sigma'}$ in this instance. Formally, *Proj* is defined by: $Proj_{\Sigma'} : \Sigma^* \to \Sigma'^*$ where

$$\begin{aligned} Proj_{\Sigma'}(\varepsilon) &:= \varepsilon \\ Proj_{\Sigma'}(e) &:= \begin{cases} e & \text{if } e \in \Sigma' \\ \varepsilon & \text{if } e \notin \Sigma' \end{cases} \\ Proj_{\Sigma'}(se) &:= Proj_{\Sigma'}(s) \cdot Proj_{\Sigma'}(e) \quad \text{for } s \in \Sigma^*,\ e \in \Sigma \end{aligned}$$

We extend the definition of projection to languages L by applying it on all words: $Proj_{\Sigma'}(L) := \{Proj_{\Sigma'}(w) \mid w \in L\}$.

This definition of projection stems from [21]. Note that by the definition of concatenation 6.14 words such as $ette \in \Sigma^*$ and $ete \in \Sigma'^*$ become equivalent after projecting away t: $Proj_{\Sigma'}(ette) = Proj_{\Sigma'}(ete) = ee$ where $t \notin \Sigma'$. We will say "project away t from Σ'^*" to denote $Proj_{\Sigma}(\Sigma')$, where $\Sigma := \Sigma' \setminus \{t\}$.

6.2.18 Simulation Relation

Simulation relation is a relation between two (S)EFAs that captures the idea of one automaton being able to mimic the behavior of another. This concept is useful for reasoning about the equivalence or mimicking capabilities of different (S)EFAs.

Definition 6.16 (Simulation relation, inspired by [22, Section 7.47])**.** Consider (S)EFAs M and M^+ such that $\Sigma_M \subseteq \Sigma_{M^+}$. A relation $Sim \subseteq X_M \times X_{M^+}$ is called a *simulation relation* from M to M^+ if for all $(x_M, x_{M^+}) \in Sim$ and for all possible transitions $x_M \xrightarrow{\sigma} x'_M$, there exists a transition $x_{M^+} \xrightarrow{\sigma} x'_{M^+}$ such that $(x'_M, x'_{M^+}) \in Sim$. Furthermore, if a predicate ϕ over the state of M is satisfied by the current state of M, it should also be satisfied in M^+: $\forall (x_M, x_{M^+}) \in Sim : x_M \models \phi \implies x_{M^+} \models \phi$. If such a relation S exists and it also holds for all initial states: $\forall x_{0,M} \in X^0_M : \exists x_{0,M^+} \in X^0_{M^+} : (x_{0,M}, x_{0,M^+}) \in \text{Sim}$, we say that M is *simulated by* M^+, written as $M \preceq M^+$.

6.3 Supervisory Control Theory

In this section we formalize key concepts from Supervisory Control Theory (SCT) [6, 9], as they are defined in CIF. Recall the synthesis process outlined in Section 4.3.2. We will describe plantification, conversion to SEFA, the four properties of supervisor synthesis (safety, controllability, non-blockingness, and maximal permissiveness), and the symbolic synthesis algorithm itself in more detail below. These concepts are essential for understanding the literature study (Section 7) and the contributions of this thesis. For a more complete explanation of synthesis we refer the reader to [9].

6.3.1 Plantification

A process called "plantification" is used to reduce the number of concepts to consider during synthesis by converting requirement automata to plants. The concept of requirement automata is eliminated by relabeling them as plants and translating the restrictions they impose to invariants. To avoid imposing restrictions in the transformed plant model, a self-loop is added for every event (σ) in the requirement's alphabet across all locations. If existing edges for an event already exist, the new self-loop includes a guard (g') that negates the disjunction of those edges' guards (g), thus $g' := \neg g$. This ensures there is always a possible transition for every event in the requirement's alphabet, and thus the plantified requirement doesn't block the event in synchronization with other automata. Additionally, to preserve the original restrictions, a state/event exclusion invariant σ *needs* g' is introduced for each added self-loop.

For example, consider event e and two edges, with guards g_1 and g_2, in location l. A self-loop is created with guard $\neg(g_1 \vee g_2)$. Additionally, a state/event exclusion requirement invariant is generated that states e *needs* $(g_1 \vee g_2)$. This invariant ensures the added self-loop is never actually enabled, as $\neg(g_1 \vee g_2) \wedge (g_1 \vee g_2)$ never holds, and thus the restrictions are maintained.

6.3.2 Conversion to SEFA

The final transformation before synthesis is the conversion of the core CIF model, modeled as EFA, into a SEFA to enable symbolic synthesis. First, the core CIF model is linearized (Definition 6.12), restructuring it into a single location where all original location information is captured using variables.

Afterwards, the linearized EFA is transformed into a SEFA (Definition 6.13). A crucial part of this transformation, with respect to the safe language (Definition 6.19), is the construction of the forbidden state predicate p_f. The construction of p_f is described in Definition 6.17 below.

Definition 6.17 (Forbidden state predicate)**.** Consider an EFA M, then p_f denotes the *forbidden state predicate* [9] which is computed during the transformation of M to a SEFA (Definition 6.13) as follows:

1. Compute p_r as the conjunction of all restrictions imposed by requirement invariants.

2. Initialize the forbidden state predicate p_f as the negation of p_r: $p_f := \neg p_r$.

3. For every integer variable x, with a defined range $[x^{min}, x^{max}]$, an implicit state requirement invariant $x_{range} := x^{min} \leq x \wedge x \leq x^{max}$ is added. This implicit invariant contributes to the forbidden state predicate: $p_f := p_f \vee \neg x_{range}$.

4. For every uncontrollable event, $\sigma \in \Sigma_u$, and every edge (g, r, σ, u) with state/event exclusion requirement σ needs p, identify the states b where the edge's guard g holds but p does not: $b := g \wedge \neg p$. States b are considered non-controllable as synthesis cannot prevent enabled uncontrollable transitions. Therefore, b is added to the forbidden states: $p_f := p_f \vee b$, concluding the construction of p_f.

6.3.3 Symbolic Synthesis Algorithm

Once all preprocessing steps on the specification are completed (recall Figure 3), the symbolic synthesis algorithm of CIF can be applied to generate a supervisor. The steps of this algorithm are outlined in Algorithm 1 below. It starts with a specification SEFA M and its forbidden state predicate p_f (Definition 6.17) as input. The output is a controlled system SEFA S that is safe, controllable, non-blocking, and maximally permissive with respect to M. S is computed by first (**step 1**) initializing the set of controllable states C as all states that are not forbidden ($\neg p_f$). This step ensures the controlled system will be safe, as all forbidden states are excluded from C. Next, the algorithm enters a loop (**steps 2-8**) that iteratively refiness C until a fixed point is reached, ensuring S is controllable and non-blocking. This loop starts by storing the current set of controllable states in C' (**step 3**). Then, it performs a backward reachability search (*BRS*, Algorithm 2) from the marked states (p_m) to identify all states that can reach a marked state in C (**step 4**). Next, a backward reachability search is performed from the non-controllable states ($\neg C$) using only uncontrollable edges (E_u) to find all states in C that can reach an unsafe or blocking state uncontrollably (**step 5**). Subsequently, these states are removed from C (**step 6**). In the last step of the loop (**step 7**), an optional forward reachability search (*FRS*, similar to *BRS* but forward) is performed from the initial states (p_0) to ensure all states in C are reachable from the initial states. This step is optional as it can influence synthesis performance positively or negatively depending on the resulting predicate sizes for C and also on the predicates manipulated during synthesis, which can be smaller or larger. This in turn affects the performance of the *BRS* computations in the next iteration(s). After reaching a fixed point ($C = C'$, **step 8**), the algorithm proceeds to update the guards of controllable edges (**steps 9-11**) such that all enabled transitions stay within the controlled system state-space. Finally, the controlled system SEFA S is constructed (**step 12**) using the updated edges and restricting the initial and marked state predicates of M to C.

Algorithm 1 Symbolic Supervisor Synthesis (*SSS*), originally from [9]

```
Input: Specification SEFA $M = (V, D, \Sigma, E, p_0, p_m)$ and forbidden states $p_f$ over $V$.
Output: Controlled-system SEFA $S$.
 1: $C \leftarrow \neg p_f$
 2: repeat
 3:     $C' \leftarrow C$
 4:     $C \leftarrow BRS(p_m, E, C)$
 5:     $B \leftarrow BRS(\neg C, E_u, true)$
 6:     $C \leftarrow \neg B$
 7:     $C \leftarrow FRS(p_0, E, C)$                                ▷ Optional step.
 8: until $C = C'$
 9: for all $e \in E$ with $\sigma \in \Sigma_c$ do
10:     $g \leftarrow g \wedge relprev(e, C)$
11: end for
12: $S \leftarrow (V, D, \Sigma, E, p_0 \wedge C, p_m \wedge C)$
```

Definition 6.18 (*Synth*)**.** Consider a specification SEFA M, and a forbidden state predicate p_f over the states of M. Then $Synth(M, p_f)$ produces the supervisor SEFA S resulting from applying the symbolic supervisory controller synthesis algorithm (Algorithm 1) on M with forbidden states p_f.

The backwards reachability search used in Algorithm 1 is outlined in Algorithm 2 below. Please note that in the current version of CIF (v10.0, December 2025) a more efficient algorithm is used for state-space exploration, namely, saturation [23]. However, for understanding the synthesis algorithm, the simpler version shown here suffices.

BRS takes a start predicate P, a set of edges E, and a restriction predicate R as input. The algorithm iteratively expands the set of states P by adding states that can reach states in P via edges in E, while ensuring that only states satisfying the restriction predicate R are considered. This is accomplished by first storing the current predicate P in P' (**step 2**), then iterating over all edges e in E and updating P by adding states that can reach P through e while restricting the result to R (**steps 3-5**). *relprevIntersection* is a function that computes $relprev \wedge R$ in one operation for efficiency. The process continues until a fixed point is reached ($P = P'$, **step 6**). The fixpoint detection in CIF is more optimized than shown here, but this version suffices for understanding the synthesis algorithm. For a more detailed explanation of the symbolic synthesis algorithm, we refer to [9].

Algorithm 2 Backward Reachability Search (*BRS*), originally from [9]

```
Input: Start predicate $P$, edges $E$, and restriction predicate $R$.
Output: The coreachable states $P'$, that is, those states that can reach states in $P$ via
    edges in $E$, restricted to states in $R$.
 1: repeat
 2:     $P' \leftarrow P$
 3:     for all $e \in E$ do
 4:         $P \leftarrow P \vee relprevIntersection(e, P, R)$
 5:     end for
 6: until $P = P'$
```

6.3.4 Safe

A controlled system is *safe* if forbidden states (Definition 6.17) are not reachable. This notion is important as it ensures that the system adheres to the safety requirements specified in the specification.

Definition 6.19 (Safe)**.** Consider a plant EFA P, a requirements EFA R, a specification $M := P \| R$, a forbidden state predicate p_f over the states of M, a supervisor $S := \mathit{Synth}(M, p_f)$, and a controlled system $C := M \| S$. Then, C is **safe** if its language is a subset of the *safe language* $L_s(M)$ where:

$$L_s(M) := \{\, w \in L(M) \mid \forall x \in \mathit{States}\Big(\ \mathit{Runs}(M, \{w\})\ \Big) :\ x \models \neg p_f \,\}$$

Synthesis ensures the controlled behavior is safe, $L(C) \subseteq L_s(M)$ [9].

The intuition is that the supervisor disables transitions to forbidden states.

6.3.5 Controllable

A controlled system is *controllable* if uncontrollable events are never blocked by the supervisor, when they were enabled in the plant. This notion is important as it forces the supervisor to propagate restrictions backwards to controllable events or initial state predicates.

Definition 6.20 (Controllable)**.** Consider a plant EFA P, a requirements EFA R, a specification $M := P \| R$, a forbidden state predicate p_f, a supervisor $S := \mathit{Synth}(M, p_f)$, and a controlled system $C := M \| S$. Then, C is **controllable** iff:

$$\{wu \mid w \in L(C), u \in \Sigma_u\} \cap L(M) \subseteq L(C)$$ [24]

The intuition is that the supervisor is not allowed to block uncontrollable events in reachable states of the controlled system C, if in a corresponding state in M it is enabled. That is, if the plant does not allow an uncontrollable transition, the supervisor also does not have to allow it, and hence we take the intersection with $L(M)$.

6.3.6 Non-blocking

A controlled system is *non-blocking* if from every reachable state, a marked state is reachable. This notion is important as it ensures that the controlled system never reaches a deadlock. Please note that the definition of non-blockingness does allow deadlocks to occur in marked states.

Definition 6.21 (Non-blocking)**.** Consider a plant EFA P, a requirements EFA R, a specification $M := P \| R$, a forbidden state predicate p_f, a supervisor $S := \mathit{Synth}(M, p_f)$, and a controlled system $C := M \| S$. Then, C is **non-blocking** iff its language equals the set of all prefixes of the marked language of M:

$$L(C) = \overline{L(C) \cap L_m(M)}$$ [6].

The intuition is that if every word $w \in L(C)$ is itself a marked word, or a prefix of one ($w \in \overline{L(C) \cap L_m(M)}$), there always exists a (possibly empty) word v such that wv is a word of the marked language: $wv \in L(C) \cap L_m(M)$. In other words, every reachable state in the controlled system is either a marked state or can reach a marked state.

6.3.7 Blocking States

We refer to states from which no marked state is reachable as "blocking states". This notion is useful in later proofs. Formally, the following definition describes a function which outputs *true* iff a state is blocking.

Definition 6.22 (Blocking function)**.** Consider a function B that takes a (S)EFA M and a state $x \in X$ as input and outputs *true* iff x is blocking, i.e., it is not possible to reach a marked state from x:

$$B(M, x) := \forall w \in \Sigma^*, \forall x' \in X : (x \xrightarrow{w} x') \implies (x' \not\models p_m).$$

In the following lemma, we will demonstrate that blocking states are not reachable in the controlled system using the non-blockingness property of synthesis (Definition 6.21).

Lemma 6.23 (Blocking states are excluded from the controlled system)**.** *Consider a specification SEFA M, a forbidden state predicate p_f, a supervisor $S := Synth(M, p_f)$, and a controlled system $C := M \| S$. Then, if a state of C is blocking, it is not reachable in C. Formally:*

$$\forall x \in States(runs(C, L(C))) : \neg B(C, x).$$

Proof. The proof is by contradiction. Assume a reachable blocking state x. Then, there must exist a word w that leads to x, i.e, $x \in States(Runs(C, w))$. Since x is a blocking state, $B(C, x)$ holds. By Definition 6.22, this means no word $v \in \Sigma^*$ exists such that $wv \in L_m(C)$. Therefore, w cannot be a prefix of any word in $L(C) \cap L_m(M)$, i.e., $w \notin \overline{L(C) \cap L_m(M)}$. However, since $w \in L(C)$, this is a contradiction to the non-blockingness property (Definition 6.21), which states: $L(C) = \overline{L(C) \cap L_m(M)}$. Thus our initial assumption, blocking state x is reachable in C, must be false. □

6.3.8 Maximally Permissive

The controlled system is *maximally permissive* means that it allows for the largest possible language while still being safe, controllable, and non-blocking. This notion is important as it ensures that the supervisor does not impose unnecessary restrictions on the system's behavior, such as disallowing all behavior.

Definition 6.24 (Maximally permissive)**.** Consider a plant EFA P, a requirements EFA R, a specification $M := P \| R$, a forbidden state predicate p_f, a supervisor $S := Synth(M, p_f)$, and a controlled system $C := M \| S$. Additionally, consider the set of all controlled systems $SCNB_C(M)$, with respect to M, such that its output is **Safe**, **Controllable**, and **Non-blocking** (as defined in their respective sections). Then, C is **maximally permissive**, also known as **minimally restrictive**, when:

$$\forall C' \in (SCNB_C(M) \setminus \{C\}) : L(C') \subset L(C).$$

The intuition is that C is maximally permissive if it allows for the largest possible language with respect to the specification M. Our definition forces C to be unique, as the language of no other controlled system can be equal or incomparable to that of C. This definition is similar to that of [25], but we slightly modified it to make the uniqueness property part of the formal definition.

6.4 Temporal Logics

In this section we provide a brief overview of the temporal logics used in this thesis: Linear Temporal Logic (LTL) and Computation Tree Logic (CTL). We start by introducing the concept of Kripke structures [26], over which LTL and CTL are semantically defined. Afterwards, we introduce useful functions on Kripke structures, before defining LTL and CTL themselves.

6.4.1 Kripke Structure

Definition 6.25 (Kripke structure)**.** A **Kripke Structure (KS)** is a directed graph and formally defined as a 4-tuple $M := (S, S_0, R, \mathcal{L})$, where:

- S: A finite set of states.
- $S_0 \subseteq S$: The set of initial states.
- $R \subseteq S \times S$: A transition relation, which must be total, meaning for every state $s \in S$, there exists a state $s' \in S$ such that $(s, s') \in R$. In other words, every state has at least one outgoing transition.
- $\mathcal{L} : S \to 2^{AP}$ is a labeling function which assigns to each state a set of atomic propositions, from AP, that hold in that state.

Definition 6.26 (Runs on Kripke structures)**.** We define a function Runs_{ks} which is similar to *Runs* (Definition 6.6), but tailored for Kripke strucutres. Instead of expecting a (S)EFA and a language as input, Runs_{ks} requires a Kripke structure K and a state $x \in S_K$ and outputs the set of all possible infinite state sequences starting from x. Formally:

$$\mathit{Runs}_{ks}(K, x) := \{x_0, x_1, \dots \in S_K \mid x_0 = x, \forall i \geq 0, (x_i, x_{i+1}) \in R\}$$

Each sequence $(x_0, x_1, \dots)$ in $\mathit{Runs}_{ks}(K, x)$ is referred to as a *run* of K from x. The i-th state in a run r is obtained by projection, $r[i] = x_i$.

Definition 6.27 (Kripke structure construction)**.** Consider a SEFA M, and a language $L \subseteq L(M)$. We define the function *KS* which constructs a Kripke Structure from M and L as follows: $KS(M, L) := (S, S_0, R, \mathcal{L})$, where:

- $S := \mathit{States}(\mathit{Runs}(M, L))$, the set of all states referenced in runs of M given the language L.
- $S_0 := S \cap X^0$, the set of all initial states.
- $R := \{(\mathit{run}[i-1], \mathit{run}[i]) \in S \times S \mid \mathit{run} \in \mathit{Runs}(M, L),\ 1 \leq i < |\mathit{run}|\}$, the transition relation restricted to transitions that occur in reachable states of M given L.
- $AP := \{(v, d) \mid v \in V_M, d \in D_v\}$, the set of atomic propositions where each element is a pair of a variable and a possible valuation.
- $\mathcal{L} : S \to 2^{AP}$ is the labeling function. For each state $s \in S$, the labeling function $\mathcal{L}(s)$ contains the atomic propositions that are *true* in that state. Specifically, it includes the pair for each variable and its current value in s: $\mathcal{L}(s) := \{(v, s[v]) \mid v \in V_M\}$.

The resulting Kripke Structure thus only includes states and transitions of M that appear in runs generated by words in L.

6.4.2 Linear Temporal Logic

We will first cover the intuition behind Linear temporal logic (LTL), after which we will formally define its syntax and semantics.

LTL is a modal logic interpreted over infinite runs of Kripke structures (Definition 6.25). As the name of LTL suggests, time is conceived as a linear progression, meaning that each moment in time has a unique possible future. In a Kripke structure this means that when there are multiple outgoing transitions from a state, each transition leads to a different possible future, but only one of these futures will actually occur in a given run. Thus each branching point in a Kripke structure results in a distinct timeline.

LTL formulas are constructed using standard boolean operators (true, false, atomic propositions a, $\neg a$, $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$) and temporal operators:

- $X\ \phi$ (Next ϕ): ϕ holds in the next state.
- $\phi_1 U \phi_2$ (ϕ_1 "until" ϕ_2): ϕ_1 holds at all points in the future until some future point where ϕ_2 holds.
- $F\ \phi$ ("eventually" / "future" ϕ): ϕ holds at some state in the future.
- $G\ \phi$ ("globally" / "always" ϕ): ϕ holds at all states in the future.

LTL formulas implicitly start with a universal path quantifier (A). This means that an LTL formula ϕ is satisfied by a KS if ϕ holds in all runs starting from an initial state of KS.

Figure 5 illustrates the semantics of the various LTL operators over a run. The states where a formula holds are indicated above each state. When no formula is indicated above a state, it means that the truth value of formulas in that state are irrelevant. Additionally, states where the LTL formula holds are colored gray. Furthermore, dotted arrows indicate a continuation of the run.

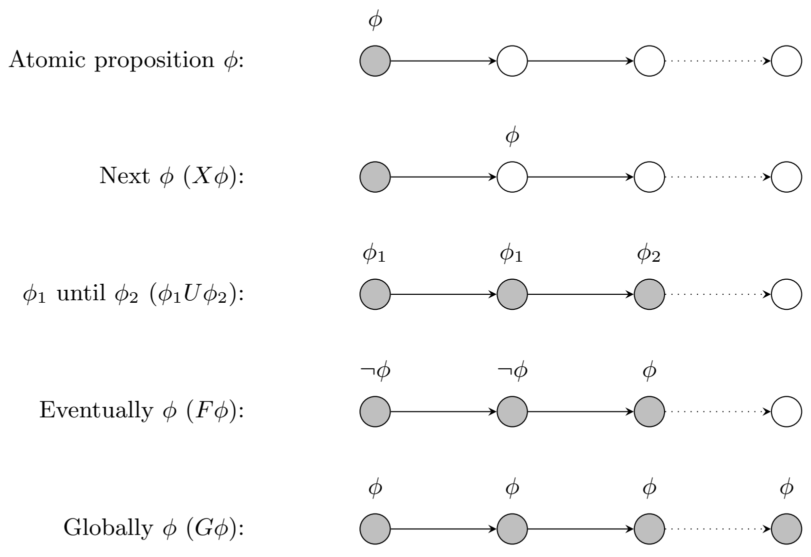

Atomic proposition ϕ:
ϕ
Next ϕ ($X\phi$):
ϕ
ϕ_1 until ϕ_2 ($\phi_1 U \phi_2$):
ϕ_1
ϕ_1
ϕ_2
Eventually ϕ ($F\phi$):
$\neg\phi$
$\neg\phi$
ϕ
Globally ϕ ($G\phi$):
ϕ
ϕ
ϕ
ϕ

Figure 5: Illustration of LTL operators over a run. This figure is inspired by [22].

Formally, LTL is defined as follows in Definition 6.28 and Definition 6.29.

Definition 6.28 (LTL Syntax)**.** LTL formulas over a set of atomic propositions AP are defined inductively:

- $true$, $false$, and every $a \in AP$ is an LTL formula.
- If ϕ_1 and ϕ_2 are LTL formulas, then so are: $\neg\phi_1$, $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $X\,\phi_1$, $F\,\phi_1$, $G\,\phi_1$, and $\phi_1\,U\,\phi_2$.

Definition 6.29 (LTL Semantics)**.** Consider a Kripke structure K (Definition 6.25) and a run $r \in Runs_{ks}(K, s)$ from some state $s \in S_K$. Note that by Definition 6.26 the length of the run r is infinite, i.e., $|r| = \infty$. The satisfaction relation $\models$ for LTL formulas is defined as follows:

- $K, r \models true$ always holds.
- $K, r \models false$ never holds.
- $K, r \models a$ iff $a \in \mathcal{L}(r[0])$, for atomic proposition a.
- $K, r \models \neg\phi$ iff $K, r \not\models \phi$.
- $K, r \models \phi_1 \wedge \phi_2$ iff $K, r \models \phi_1$ and $K, r \models \phi_2$.
- $K, r \models \phi_1 \vee \phi_2$ iff $K, r \models \phi_1$ or $K, r \models \phi_2$.
- $K, r \models X\phi$ iff $K, r[1\ldots] \models \phi$.
- $K, r \models \phi_1 U \phi_2$ iff there exists $j \geq 0$ such that $K, r[j\ldots] \models \phi_2$ and for all $0 \leq k < j$, $K, r[k\ldots] \models \phi_1$.
- $K, r \models F\phi$ iff there exists $j \geq 0$ such that $K, r[j\ldots] \models \phi$.
- $K, r \models G\phi$ iff for all $j \geq 0$, $K, r[j\ldots] \models \phi$.

We extend the satisfaction relation to states in K as follows: for state $s \in S_K$, $K, s \models \phi$ iff for all runs $r \in Runs_{ks}(K, s)$, $K, r \models \phi$. Furthermore, we extend the satisfaction relation to Kripke structures as a whole: $K \models \phi$ iff for all initial states $s_0 \in S_0$, $K, s_0 \models \phi$.

6.4.3 Computation Tree Logic

We will first cover the intuition behind Computation Tree Logic (CTL), after which we will formally define its syntax and semantics.

CTL is a branching-time logic, which means that each moment in time may diverge into various possible futures. This property stems from the fact that CTL formulas are interpreted over infinite computation trees, which describe all possible behaviors of a Kripke structure. Every transition taken can be seen as a decision, i.e., a branch in the computation tree. CTL supports the same temporal and boolean operators as LTL, but all temporal operators must be directly preceded with a path quantifier: A (every infinite path from this state) or E (there exists an infinite path from this state). Examples of combinations are AXϕ: ϕ holds in all possible next states, EXϕ: there exists a possible next state where ϕ holds. Figure 6.4.3 illustrates the semantics of several CTL operators over a computation tree of a Kripke structure K. Dashed lines indicate continuation of the computation tree, and ϕ above a node indicates that a formula ϕ holds in that state. When ϕ is not present above a node, it means that the truth value of ϕ is irrelevant. An s inside a node indicates it is the state from which the computation tree is constructed.

Formally, CTL is defined as follows in Definition 6.30 and Definition 6.31.

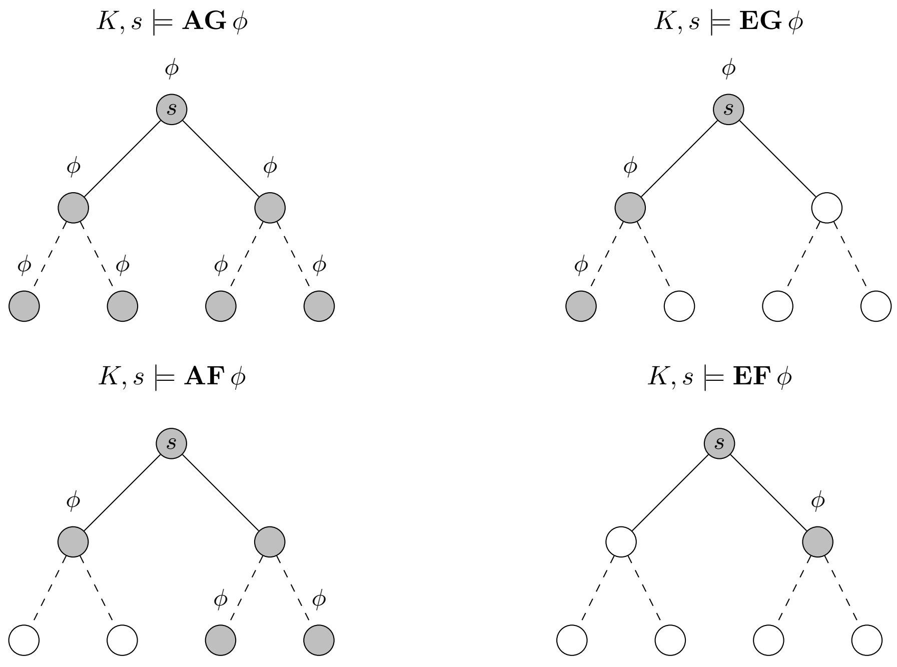

$K, s \models \mathbf{AG}\,\phi$
$K, s \models \mathbf{EG}\,\phi$
$K, s \models \mathbf{AF}\,\phi$
$K, s \models \mathbf{EF}\,\phi$
s
ϕ

Figure 6: Illustration of CTL operators over a computation tree. This figure is inspired by [27].

Definition 6.30 (CTL Syntax)**.** CTL formulas over a set of atomic propositions AP are defined inductively:

- $true$, $false$, and every $a \in AP$ is a CTL formula.
- If ϕ_1 and ϕ_2 are CTL formulas, then so are: $\neg\phi_1$, $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $AX\,\phi_1$, $EX\,\phi_1$, $AG\,\phi_1$, $EG\,\phi_1$, $AF\,\phi_1$, $EF\,\phi_1$, $A(\phi_1\,U\,\phi_2)$, and $E(\phi_1\,U\,\phi_2)$.

Definition 6.31 (CTL Semantics)**.** Consider a Kripke structure K (Definition 6.25) and a state $s \in S_K$. The satisfaction relation $\models$ for CTL formulas is defined as follows:

- $K, s \models true$ always holds.
- $K, s \models false$ never holds.
- $K, s \models a$ iff $a \in \mathcal{L}(s)$, for atomic proposition a.
- $K, s \models \neg\phi$ iff $K, s \not\models \phi$.
- $K, s \models \phi_1 \wedge \phi_2$ iff $K, s \models \phi_1$ and $K, s \models \phi_2$.
- $K, s \models \phi_1 \vee \phi_2$ iff $K, s \models \phi_1$ or $K, s \models \phi_2$.
- $K, s \models AX\,\phi$ iff for all $r \in \mathit{Runs}_{ks}(K, s)$, $K, r[1] \models \phi$.
- $K, s \models EX\,\phi$ iff there exists $r \in \mathit{Runs}_{ks}(K, s)$ such that $K, r[1] \models \phi$.
- $K, s \models AG\,\phi$ iff for all $r \in \mathit{Runs}_{ks}(K, s)$ and for all $j \geq 0$, $K, r[j] \models \phi$.
- $K, s \models EG\,\phi$ iff there exists $r \in \mathit{Runs}_{ks}(K, s)$ such that for all $j \geq 0$, $K, r[j] \models \phi$.
- $K, s \models AF\,\phi$ iff for all $r \in \mathit{Runs}_{ks}(K, s)$, there exists $j \geq 0$ such that $K, r[j] \models \phi$.

- $K, s \models EF\,\phi$ iff there exists $r \in \mathit{Runs}_{ks}(K, s)$ and $j \geq 0$ such that $K, r[j] \models \phi$.
- $K, s \models A(\phi_1\,U\,\phi_2)$ iff for all $r \in \mathit{Runs}_{ks}(K, s)$, there exists $j \geq 0$ such that $K, r[j] \models \phi_2$ and for all $0 \leq k < j$, $K, r[k] \models \phi_1$.
- $K, s \models E(\phi_1\,U\,\phi_2)$ iff there exists $r \in \mathit{Runs}_{ks}(K, s)$ and $j \geq 0$ such that $K, r[j] \models \phi_2$ and for all $0 \leq k < j$, $K, r[k] \models \phi_1$.

We extend the satisfaction relation to Kripke structures as a whole: $K \models \phi$ iff for all initial states $s_0 \in S_0$, $K, s_0 \models \phi$.

7 Literature Study

In this section, we present a literature study on liveness specification in supervisor synthesis within the RW-framework [6]. Specifically, we investigate existing approaches for specifying liveness requirements that are compatible with CIF and are applicable to the RWS case study (Section 5). We start, in Section 7.1, by outlining the methodology of the literature study, after which we describe, in Section 7.2, the selection process using the outlined methodology. Next, we describe in Section 7.3 the evaluation process used to analyze the selected papers. Subsequently, we present the evaluation of the selected papers in Sections 7.4 to 7.8. Finally, we provide a summary of our findings in Section 7.9.

7.1 Methodology

From the analysis of the RWS case study (Section 5), it became clear that CIF's current liveness specification capabilities (marked states) are insufficient to solve the challenge of RWS. Therefore, the goal of this literature study is to identify and analyze existing approaches for specifying liveness requirements in supervisor synthesis within the RW-framework [6]. Furthermore, we are specifically interested in approaches that can express the liveness requirement of RWS, which we define precisely in Section 7.3.

Concretely, we investigate existing liveness specification formalisms for supervisory control theory that:

- Are compatible with CIF and the RW-framework in terms of modeling constructs and synthesis properties.
- Can express the liveness requirement of RWS, i.e., from every reachable state, multiple marked states can be reached in the controlled system.
- Are simple to express and understand for users of CIF.

We aim to achieve this goal by conducting a literature study of relevant recent papers. As a starting point, we use *Offline supervisory control synthesis: taxonomy and recent developments* [28], where articles from 2017 up to and including 2022 are considered. Five sources on requirements in temporal logic were published during this period. These form the initial sources for our literature study.

To cover more recent work, we performed an additional search to fill the gap between 2022 and July 2025 (the time this literature study is conducted). In contrast to [28], we restricted this search to papers that reference the RW-framework [6]. Within this collection of papers, we searched for the keywords *liveness*, *LTL*, and *CTL* for the years 2023 to and including 2025. We only search for LTL and CTL as they are simple to understand while powerful enough to express the liveness requirement needed by RWS, which we will further discuss in Section 7.3.

7.2 Selection Process

To ensure that the approaches we evaluate are comparable and relevant to CIF, we apply the following inclusion criteria:

- The work is formulated within, or directly compatible with, the RW-framework [6]
- Specification modeling is comparable to the formalisms used in CIF and
- The work discusses liveness or temporal logic specifications in the context of supervisor synthesis.

We exclude approaches that significantly alter the four properties of RW-synthesis (safety, controllability, non-blockingness, and maximal permissiveness) or that are formulated in the reactive synthesis (game-theoretic) framework [29]. The reactive synthesis framework is outside the scope of this literature study, as it generally does not align with the RW-framework [6] due to different terminology and, more importantly, different definitions for safety, controllability, non-blockingness, and specifically maximal permissiveness [30]. Reactive synthesis typically aims at a winning strategy, instead of a unique maximally permissive supervisor (Definition 6.24).

In the section "Requirements in temporal logic" of [28], the first sources mentioned are [31] and [32]. These papers are excluded from further evaluation, as their definition of maximally permissive synthesis does not align with CIF's. As mentioned by the authors of [28], for [31] "it was explained how all maximally permissive solutions can be generated", and for [32] "there can be multiple maximally permissive solutions". Both cases clearly violate the uniqueness of a maximally permissive supervisor as defined in Definition 6.24, which we aim to preserve.

Afterwards, the source [33] is mentioned, which we include in our evaluation as it adds support for a subset of CTL while preserving the uniqueness of the maximally permissive supervisor. Subsequently, [34] is mentioned, which we also include, as it adds support for a subset of LTL. The authors of [28] do not state which synthesis properties are maintained by the approach of [34], so we evaluate this ourselves. Lastly, [35] is mentioned, which we also include, as it similarly adds support for LTL while no synthesis properties are discussed in [28]. In summary, from the taxonomy by [28] we include [33], [34], and [35] in our evaluation.

Next, we performed an additional search to fill the gap between 2022 and July 2025. We queried Google Scholar using "liveness *OR* LTL *OR* CTL" for papers citing the RW-framework [6]. This search yielded 84 papers, of which we excluded 77 based on their titles, as they appeared to be irrelevant to our goal. For the remaining seven papers we screened the abstracts and conclusions, after which we excluded four more papers: [36], [37], [38], and [39], as they were not in the RW-framework but in the reactive synthesis framework [29].

Additionally, we excluded [40], as it explicitly states that liveness requirements are considered future work. Furthermore, we excluded [41], as it works with probabilistic models, which does not align with the synthesis models considered in CIF. Lastly, we exclude [42], as it uses decentralized synthesis whereas CIF uses monolithic synthesis. Nevertheless, [42] is the decentralized version of [35], which we include in our evaluation. Therefore, the approach of [42] is still indirectly covered in our evaluation.

Based on an examination of relevant literature cited by the four selected sources, we additionally include [13] (referenced by [33] and [35]) and [43] (referenced by [35]). We include [13], as it appears to be a foundational paper in liveness specifications for supervisor synthesis. Whereas we include [43] as it presents a different (model-based) approach to temporal specifications in supervisor synthesis. This work only addresses safety properties (no liveness). Nevertheless, we consider it valuable as it provides an alternative perspective on temporal specifications in supervisor synthesis.

In total, we evaluate the following five papers in this literature study: [13], [33], [35], [34], and [43].

7.3 Evaluation Process

In the preliminaries (Section 6), we formally defined the constructs and properties used in CIF. These constructs and properties serve as the starting point for our evaluation of the selected papers. Additionally, we require a precise formalization of the liveness requirement based on the RWS case study (Section 5).

Informally, RWS requires that, in the controlled system, it must always be possible to reach a state in which a waterway lock gate is open, and likewise a state in which it is closed. Now, consider *open* and *closed* to be state predicates that hold in exactly those states in which a waterway lock gate is open and closed, respectively. Then, the liveness requirement which RWS needs to hold in the controlled system can be precisely captured using the CTL formula $(AG\ EF\ open) \wedge (AG\ EF\ closed)$ (when the controlled system is total, Definition 6.8), by the definition of CTL semantics (Definition 6.30). This formula states that, from every reachable state, it must always be possible to eventually reach a state satisfying *open*, and likewise it must always be possible to eventually reach a state satisfying *closed*.

Based on the CIF concepts from Section 6 and the RWS liveness requirement defined above, we formulate four guiding questions to evaluate the selected papers:

1. How does the specification formalism used in the paper compare to EFAs in CIF (in terms of states, events, variables, guards, marked states, and initial states)?

2. Do the definitions of safety, controllability, non-blockingness, and maximal permissiveness align with Definitions 6.19, 6.20, 6.21, and 6.24, respectively?

3. Which liveness requirements can be specified, and in particular, can the approach express requirements equivalent to $(AG\ EF\ open) \wedge (AG\ EF\ closed)$?

4. What are the limitations of the approach?

The first and second questions evaluate the compatibility of the approach with CIF and the RW-framework [6] in general. The third question evaluates whether, and to what extent, the approach can express the liveness requirement of RWS as defined above. The fourth question evaluates the limitations of the approach, both in general and specifically in the context of CIF and RWS.

7.4 Multitasking Supervisory Controller Synthesis

We begin the evaluation with [13], as it is a foundational paper in the field of liveness specifications in supervisor synthesis. In the literature, marked states are also known as *tasks* which a controlled system can complete by reaching a marked state. [13] introduces the concept of *multitasking supervisory control*, where the goal is to ensure that multiple tasks can be completed in the controlled system, i.e., to ensure the reachability of multiple marked states. The authors also refer to this approach as *coloring*, and tasks/marked states as states of different colors. In the following subsections, we evaluate [13] based on the four guiding questions outlined in Section 7.3.

7.4.1 Specification Formalism Comparison

The authors use a Colored Marking Generator (CMG) to model a specification, where a CMG is formally defined as a 6-tuple $G := (Q, \Sigma, C, E, \gamma, q_0)$, where:

- Q: The finite set of states.
- Σ: a finite set of events (the alphabet), partitioned into two disjoint sets: Σ_c for controllable events and Σ_u for uncontrollable events, i.e., $\Sigma := \Sigma_c \cup \Sigma_u$, $\Sigma_c \cap \Sigma_u = \emptyset$.
- C is a finite set of colors, representing all the classes of tasks the system can execute.
- $E : Q \times \Sigma \to Q$ is the state transition function.
- $\gamma : Q \to \mathcal{P}(C)$ is the marking function, which assigns to each state of Q a (possibly empty) subset of colors.
- q_0 is the initial state.

A CMG is notably different from an EFA in CIF (Definition 6.1), as it does not include variables and consequently neither guards nor error predicates nor assignments on transitions. Furthermore, there are no marked states in a CMG, instead, the marking function γ assigns colors to states. Also notably different is that q_0 is a single initial state, whereas an EFA in CIF can have multiple initial states.

The absence of variables can be bridged by modeling each combination of variable valuations as separate states in the CMG, as we will see in the work of [33]. The restriction to a single initial state can be overcome by introducing a new initial state with controllable transitions to the desired initial states. This way, the restrictions CIF imposes on initial states can be applied to the added controllable transitions instead.

7.4.2 Synthesis Properties Comparison

The authors of [13] define safety, controllability, and maximal permissiveness similarly to classical supervisory control theory [6], which forms the basis of CIF's definitions (Definitions 6.19, 6.20, and 6.24, respectively). Non-blockingness is defined differently, as it is defined with respect to colors instead of marked states.

Safety is defined such that the controlled system only allows sequences of events within the admissible language of the specification. Controllability is defined such that the supervisor cannot disable uncontrollable events. Unlike in CIF, restrictions cannot be imposed on the initial state predicate, as there is only a single initial state in a CMG. But when multiple initial states are modeled using a new initial state with controllable transitions to the desired initial states (as mentioned above), the restrictions that are in CIF imposed on the initial state predicates are applied on the added controllable transitions instead. Non-blockingness is defined such that from every reachable state, for each color in C, there exists a path leading to a state marked with that color. The authors mention that classical non-blockingness can be obtained as a special case by considering a single color. Maximal permissiveness is defined such that the supervisor allows all behaviors that do not violate safety, controllability, and non-blockingness.

7.4.3 Liveness Specification Capabilities

As discussed above, the approach of [13] enables specifying multiple states (tasks) that must remain reachable in the controlled system. The authors do not directly tie this capability to any temporal logic formalism, but [33] does connect multitasking supervisory control to CTL formula $AG\ EF\ c$ for all colors c. However, [33] does not formally define this connection. Furthermore, we observe that this connection is not fully aligned with CTL semantics (Definition 6.30), as when there is just a single color (or one state marked with all colors), then the controlled system is not necessarily total (Definition 6.8). This holds because [13] defines non-blockingness, when there is only a single color, equal to classical RW-synthesis and RW-synthesis allows livelocks in marked states [6]. Nevertheless, we accept multitasking supervisory control as a way to express the liveness requirement of RWS.

7.4.4 Limitations

The authors mention scalability as a limitation, not specifically in the context of liveness specifications, but in general for supervisor synthesis. As an example, they mention that the specification of their case study contained 8.13×10^5 states. The work was published in 2005, and since then significant advancements have been made in supervisor synthesis to improve performance. Consequently, 8.13×10^5 states is not necessarily large by today's standards. However, the state-space size is not the only factor influencing performance, the structure of a specification also plays a role. This point can be illustrated using two models from the CIF benchmark. CIF includes a set of 23 benchmark models stemming from, or inspired by, industry, government, and academia for supervisor synthesis, of which 13 we describe in Section 10.1.1. From the 13 CIF benchmarks, consider the wafer-scanner- and waterway lock model. Their uncontrolled system state-space sizes are 5.30×10^5 and 5.96×10^{32} states, respectively (according to [9]). However, the wafer-scanner model requires 3.47×10^8 Binary Decision Diagram (BDD) [44] operations to synthesize whereas the waterway lock model requires only 4.75×10^4 operations (when using CIF v8.0). This example illustrates that state-space size is not the only factor influencing synthesis performance.

From the perspective of CIF, we mainly see limitations regarding the expressiveness and convenience of defining specifications. Specifically, CMGs lack variables and multiple initial states, which limits the convenience of specifying complex systems compared to CIF's capabilities. But this limitation is not consequential for the adoption of multitasking supervisory control in CIF.

In summary, [13] provides a way to express multiple reachability requirements via colors and extends classical RW-synthesis. Based on our evaluation, we consider the essence (coloring) of this approach to be compatible with CIF.

7.5 Multiple Reachability Requirements in Supervisory Control

The work [33] builds on [13] and uses it as a foundation for more complex (liveness) specifications. Specifically, [33] extends the approach of [13] to additionally support invariance. In the subsequent subsections we evaluate [33] based on the four guiding questions outlined in Section 7.3.

7.5.1 Specification Formalism Comparison

The authors use a Labeled Transition System (LTS) to model a specification, where they define an LTS as a 3-tuple $\mathcal{L} = (Q, \Sigma, E)$ where:

- Q: a finite set of states. The state is defined by a set of state variables, so that each state $q \in Q$ corresponds to a valuation of these variables.
- Σ: a finite set of events (the alphabet), partitioned into two disjoint sets: Σ_c for controllable events and Σ_u for uncontrollable events, i.e., $\Sigma := \Sigma_c \cup \Sigma_u$, $\Sigma_c \cap \Sigma_u = \emptyset$. The input is represented by a set of input variables, so that relational expressions involving those input variables define subsets of Σ.
- $E \subseteq Q \times \Sigma \times Q$: a set of transitions, where each transition (q, σ, q^+) indicates that the system moves from state q to state q^+ given input σ.

The authors require the LTS $\mathcal{L}$ to be *deterministic*, meaning every pair (q, σ) corresponds to at most one transition $(q, \sigma, q^+) \in E$.

The most basic properties of states are given by atomic propositions. These are expressions using the state variables. For example, if q is a state variable with domain $\{1, 2, \dots, 7\}$, then expressions such as:

$$(q = 4), \quad (q \in \{3, 4\}), \quad (q \neq 5)$$

are atomic propositions that evaluate to true or false for each state in Q.

Additionally, the authors expand the specification formalism to include Combined Invariance and Reachability (CIR) using a subset of CTL, where CIR is formally defined as follows:

$$AG\left(\bigwedge_{i\in I} p_i \;\wedge\; \bigwedge_{j\in J} EF(p_j)\right)$$

Both p_i and p_j are boolean predicates over states, thus they do not contain any CTL operators. CIR combines:

- **Invariance requirements** $AG(p_i)$: property p_i must always hold (safety).
- **Reachability requirements** $AG(EF(p_j))$: from every reachable state, it must always be possible to eventually reach a state satisfying p_j (non-blocking / liveness).

The LTS formalism is similar to an EFA in CIF (Definition 6.1), as both include variables, guards (relational expressions involving variables), and (implicit in the case of LTS) assignments on transitions. This work also covers the convenience of invariance, which CIF supports as well. However, there are some notable differences. Firstly, there are no marked states in an LTS, where marked states are fundamental in CIF. The concept of marked states can be mimicked similarly to [13] by using reachability requirements. Secondly, and most notably, there is no initial state predicate in an LTS. We will come back to this in the next subsection.

Overall, the LTS formalism appears to be largely compatible with EFAs in CIF, with some limitations due to the lack of explicit marked states and initial state predicates.

7.5.2 Synthesis Properties Comparison

The authors explicitly link their work to classical supervisory control theory [6], which consequently connects it to CIF.

Safety requirements are defined using the invariance operator AG, ensuring that the controlled system only allows sequences of events within the admissible language of the specification. Controllability is defined such that the supervisor cannot disable uncontrollable events. Non-blockingness is defined using the reachability operator EF, ensuring that from every reachable state there exists a path leading to a state satisfying the reachability requirement. Maximal permissiveness is defined differently compared to classical supervisory control theory: the authors consider a solution to be maximal if it allows the largest possible set of states. Using this definition, initial states can be mimicked by choosing a set of states for which at least one must remain in the maximal state-space. When no intended initial states are present in the maximal state-space, the result is considered to be the empty supervisor. Additionally, as previously mentioned, the reachability operator EF can be used to define marked states. Utilizing both initial- and marked state mimicking combined, the definition of maximally permissiveness can be made compatible to the one used in CIF (Definition 6.24).

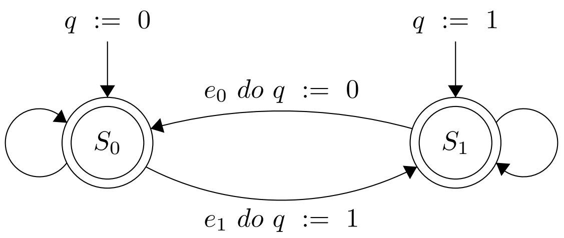


Figure 7: An example EFA where multiple maximally permissive supervisors exist for $AG(EF(AG(q = 1) \vee AG(q = 0)))$. This figure is inspired by [33].

7.5.3 Liveness Specification Capabilities

The capabilities for specifying liveness requirements are similar to [13], as the possible reachability of multiple desired states can be defined. Since the approach of [33] extends the modeling formalism to include variables, more complex liveness requirements can be specified than with CMGs. The liveness requirement of RWS, as defined in terms of CTL, can be specified within the subset of CTL formulas supported by CIR.

7.5.4 Limitations

The authors mention the requirement of the LTS to be deterministic as a limitation. This limitation is relevant as CIF does support non-determinism only for uncontrollable events in the context of symbolic synthesis. Furthermore, the authors mention that their solution is memoryless, meaning it does not track past states, this limits the expressiveness of CIR specifications. This limitation is not relevant in the context of CIF, as CIF also does not track past states. Another limitation mentioned by the authors is their limited support for CTL, as only invariance and reachability are supported. This limitation is relevant in the context of CIF, as it limits the expressiveness of specifications, but in the context of the RWS case study, this limitation is not problematic, since only reachability requirements are needed.

The authors also mention fundamental limitations regarding CTL specifications for RW-synthesis in general. Allowing full CTL can lead to incomparable solutions of the synthesis problem, violating uniqueness in the definition of maximally permissive (Definition 6.24). Therefore, the authors restrict their approach to CIR specifications to ensure that the solution is unique. Consider, for example, the EFA depicted in Figure 7. This EFA has two (initial) locations S_0 and S_1, and a boolean variable q which can be either 0 or 1. There are two controllable events e_0 and e_1 which set q to 0 and 1, respectively. The liveness requirement $AG(EF(AG(q = 1) \vee AG(q = 0)))$ states that from every reachable state it must always be possible to eventually reach a state where either q remains 1 forever, or q remains 0 forever. There are two maximally permissive solutions: never allowing e_1 or never allowing e_0. The solution has to disable one of the events, as otherwise there is no guarantee that either $AG(q = 1)$ or $AG(q = 0)$ will eventually hold. Since the requirement states that there must exist a path to either $AG(q = 1)$ or $AG(q = 0)$ from every reachable state, the initial value of q does not matter.

In summary, [33] extends [13] by adding support for variables and invariance, enabling more complex liveness specifications while largely preserving compatibility with CIF.

7.6 Synthesis with Canonical LTL and Fair Events

The work [35] introduces an approach for supervisor synthesis using canonical LTL requirements, alongside plants modeled as Discrete Event Systems (DES). Canonical LTL is a version of LTL such that safety requirements and liveness requirements are defined over past LTL formulas. Their approach utilizes fairness assumptions on events to ensure constant progress towards marked states. In the subsequent subsections we evaluate [35] based on the four guiding questions outlined in Section 7.3.

7.6.1 Specification Formalism Comparison

The authors use a discrete event system (DES) to model a specification, where they define a DES formally as a 5-tuple $\mathcal{G} = (V, Q, \Sigma, E, p_0)$ where:

- V: A finite set of typed state variables. Each state variable $v \in V$ has an associated domain Range(v).
- Q: The finite set of states, defined as the cross product of the ranges of all state variables: $Q := \prod_{v \in V} \text{Range}(v)$. Each state $q \in Q$ is a valuation of all state variables.
- Σ: a finite set of events (the alphabet), partitioned into two disjoint sets: Σ_c for controllable events and Σ_u for uncontrollable events, i.e., $\Sigma := \Sigma_c \cup \Sigma_u$, $\Sigma_c \cap \Sigma_u = \emptyset$.
- $E : \Sigma \times Q \to Q$: A (deterministic) partial state transition function. For each state $q \in Q$, the function $E(\sigma, q)$ is defined only for a subset of Σ, indicating which events are enabled at that state.
- p_0: Predicate over Q representing the initial states.

Furthermore, a specification of a plant may select a subset of uncontrollable events $\Sigma_f \subseteq \Sigma_u$ as *fair* events. The fair event set itself is also partitioned into two disjoint sets: $\Sigma_f = \Sigma_C \cup \Sigma_J$ such that $\Sigma_C \cap \Sigma_J = \emptyset$. Σ_C denotes *strongly fair events*, whereas Σ_J denotes *weakly fair events*. When in a run of a DES a strongly fair event $\sigma \in \Sigma_C$ is enabled infinitely often, then σ must be executed infinitely often. When in a run of a DES a weakly fair event $\sigma \in \Sigma_J$ is enabled continuously from some point onward, then σ must be executed infinitely often. We will illustrate the purpose of fair events in the EFA depicted in Figure 8. Consider event e_1 to be an uncontrollable event. If e_1 is not considered to be a fair event, then there is no guarantee that e_1 will ever be executed and consequently the marked state may never be reached. However, if e_1 is considered to be a fair event (both fairness types suffice), then e_1 is assumed to be executed at some point, and consequently the marked state will eventually be reached. Fair events can be considered uncontrollable events that are part of regular operation of the system, while non-fair uncontrollable events are considered to be faults or rare events. For example, a fair uncontrollable event in a waterway lock may be a sensor detecting whether a lock gate is closed, while a non-fair uncontrollable event may be an emergency stop button being pressed.

While the plant is modeled as DES, requirements are modeled using *canonical LTL* formulas over the states of the DES. The authors define canonical LTL as:

$$G\left(P \cdot \bigwedge_{i \in I} F(M_i)\right)$$

Where P and M_i are past LTL formulas over states in Q. Past formulas included operators for: has always been, once, previously, and since. Canonical LTL formulas are used to approximate marked states (through M_i) and safety requirements (through P).

The DES formalism is similar to a SEFA in CIF (Definition 6.2), except SEFAs in CIF include explicit guards while in a DES this is captured through the partial state transition function. Furthermore, both formalisms include invariance (safety). Marked states are defined differently, which we will discuss in the non-blockingness definition of next subsection.

7.6.2 Synthesis Properties Comparison

The authors draw explicit commonalities and distinctions between their definitions and classical supervisory control theory [6]. Safety is defined such that nothing "bad" may happen, where it is formally defined using the invariance operator G in LTL. Controllability is defined similarly to classical supervisory control theory [6], such that the supervisor cannot disable uncontrollable events. However, in contrast to CIF and [6], the authors include fairness assumptions on a subset of uncontrollable events. Non-blockingness is most notably defined differently, as it is defined such that all marked states / conditions must be visited / met infinitely often. Maximal permissiveness is also defined differently, due to the different definition of non-blockingness. A supervisor is maximally permissive if it allows the largest possible set of behaviors while ensuring safety, controllability, and constant progress towards marked states.

The difference in non-blockingness (and consequent difference in maximal permissiveness) compared to CIF is significant. This difference is fundamental due to the contrasting views of liveness from an LTL and RW-framework perspective. We will elaborate on this difference further in Section 7.6.4.

7.6.3 Liveness Specification Capabilities

Canonical LTL allows specifying that marked states must be visited infinitely often by always making progress to a different marked state. This liveness requirement is stricter than the one required by RWS, as defined in Section 7.3. In terms of a waterway lock, the liveness requirement specifiable in canonical LTL is that a waterway lock gate must open and close infinitely often. Such that when, for example, a waterway lock gate is open, all actions must contribute to eventually closing that waterway lock gate again. In the subsequent subsection we will discuss how such requirements drastically limits possible behavior in the controlled system.

7.6.4 Limitations

Regarding limitations, the authors created a purely theoretical framework and did not implement their approach. This is not necessarily a limitation regarding the applicability to CIF.

The most significant limitation from the perspective of CIF is the different definition of non-blockingness. In CIF, non-blockingness ensures that always a marked state remains reachable, while in [35], non-blockingness ensures that marked states are visited infinitely often. As mentioned previously, this difference is fundamental due to the contrasting views of liveness from an LTL and RW-framework perspective. Consider the (canonical) LTL requirement $G(F(S_1))$ specified for the EFA depicted in Figure 8,

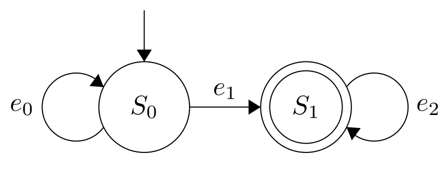

$$\begin{tikzcd} {} \arrow[d] & \\ S_0 \arrow[loop left, "e_0"] \arrow[r, "e_1"] & \boxed{S_1} \arrow[loop right, "e_2"] \end{tikzcd}$$

Figure 8: An example EFA where the (canonical) liveness requirement $G(F(S_1))$ drastically limits possible behavior.

where all events (e_0, e_1, and e_2) are considered to be controllable. Now consider maximal permissiveness as defined in Definition 6.24. By this definition, a supervisor is considered maximally permissive if it allows the largest possible language while ensuring safety, controllability, and non-blockingness. To ensure the liveness requirement $G(F(S_1))$ is enforced, the supervisor must disable event e_0 at some point, as otherwise there is no guarantee that state S_1 will be visited at all. But for all possible supervisors that ensure $G(F(S_1))$ in this scenario, there exists another supervisor that allows for one more self-loop on S_0 before disabling e_0. Consequently, no maximally permissive supervisor exists in this case. The authors solve this problem by requiring that the supervisor ensures progress to marked states, thus consequently e_0 must be disabled immediately. This example illustrates how the definition of non-blockingness in [35] can drastically limit possible behavior in the controlled system compared to CIF's definition.

In summary, while [35] provides a way to express liveness requirements using LTL, its definition of non-blockingness and maximal permissiveness differ significantly from CIF's, making it less suitable for our purpose. Furthermore, this difference drastically limits possible behavior in the controlled system. Lastly, the LTL liveness requirements are stricter than the one required by RWS.

7.7 LTL$_{\backslash 0}$ Synthesis Utilizing Abstraction

The work of [34] introduces an approach for supervisor synthesis using a subset of LTL called LTL$_{\backslash 0}$. This subset excludes the next operator X. By doing so, the authors utilize an abstraction technique called *stutter bisimulation abstraction*, which enables more efficient synthesis. In the subsequent subsections we evaluate [34] based on the four guiding questions outlined in Section 7.3.

7.7.1 Specification Formalism Comparison

The authors use a transition system (TS) to model a specification, where they formally define a TS as a 5-tuple $\mathcal{G} = (Q, Q_0, E, \mathcal{P}, \models)$ where:

- Q: The finite set of states.
- $Q_0 \subseteq Q$: The set of initial states.
- $E \subseteq Q \times Q$: The state transition relation.
- $\mathcal{P}$: A set of atomic propositions.
- $\models$: A satisfaction relation between states and atomic propositions.

Additionally, requirements can be modeled using LTL$_{\backslash 0}$ formulas over the atomic propositions in $\mathcal{P}$. LTL$_{\backslash 0}$ is defined similarly to standard LTL, but excludes the next operator X. This exclusion enables the use of stutter bisimulation abstraction [22]. Stutter bisimulation abstraction groups states first on their valuations of atomic propositions, and then merges states that have the same transitions to other groups of states. We will explain how this abstraction is used in Section 7.7.4.

The TS formalism is in many aspects different from an EFA in CIF (Definition 6.1). Firstly, there are no variables in a TS, and consequently neither guards nor assignments on transitions. As mentioned previously, this limitation can be overcome by modeling each combination of variable valuations as separate states in the TS. Secondly, there are no marked states in a TS, where marked states are fundamental in CIF. Marked states can be mimicked by introducing an atomic proposition M and LTL$_{\backslash 0}$ formula $G(F(M))$. But note by the Semantics of LTL (Definition 6.29), this formula requires that states satisfying M are visited infinitely often. This is a significant difference compared to marked states in CIF, which only require the existence of a path to a marked state. Lastly, and most notably, there is no concept of events in a TS. Events are fundamental in supervisor synthesis, as all definitions are based on languages, for which events are the building blocks.

7.7.2 Synthesis Properties Comparison

With respect to safety, controllability, non-blockingness, and maximal permissiveness, the authors do not explicitly define these properties in their work. However, we can infer their definitions based on the context of supervisor synthesis. Safety can be inferred to be defined such that the controlled system only satisfies the safety requirements specified in $\mathrm{LTL}_{\setminus 0}$. Controllability is undefined as there are no events in a TS. Non-blockingness can be inferred to be defined such that from every reachable state, states satisfying a marked state condition (e.g., M) can be reached infinitely often. The authors additionally require the controlled system to be deadlock-free, meaning that from every reachable state there exists at least one outgoing transition. Regarding maximal permissiveness, the authors accept a supervisory controller when it ensures deadlock-freeness and adheres to $\mathrm{LTL}_{\setminus 0}$ requirements. However, there is no notion of maximal permissiveness as defined in Definition 6.24.

7.7.3 Liveness Specification Capabilities

The capabilities for specifying liveness requirements are similar to [35], as $\mathrm{LTL}_{\setminus 0}$ requirements can specify that states satisfying a marked state condition (e.g., M) must be visited infinitely often. This liveness requirement is, likewise, stricter than the one required by RWS, as defined in Section 7.3.

7.7.4 Limitations

The authors mention as limitation that their approach is restricted to $\mathrm{LTL}_{\setminus 0}$, but this enables the strength of their approach, which is the use of stutter bisimulation abstraction. However, this abstraction comes at the cost of another limitation, which is the runtime overhead of supervisory controller execution. The supervisor must consult runtime history and the current state to determine whether it must transition to another state group or not. Synthesis using regular bisimulation abstraction [45] offers a controller which maps control decisions "significantly faster", according to the authors. However, no concrete performance differences are provided.

Such overhead in supervisory controller execution is not present in the context of CIF. Furthermore, as mentioned previously, CIF currently does not keep track of past states either. The biggest limitation regarding applicability to CIF is the lack of events in a TS. Events are not only fundamental in CIF, but to supervisor synthesis in RW-framework in general. As all definitions are based on languages, for which events are the building blocks.

Regarding the limitation of LTL specifications for synthesis in general, as discussed in Section 7.6.4, the authors overcome this by, likewise, forcing progress to states which require such due to LTL liveness requirements (e.g. $G(F(\phi))$).

In summary, while [34] introduces an interesting abstraction technique for LTL synthesis, its lack of events and differing definitions of non-blockingness and maximal permissiveness make it less suitable for our purpose.

7.8 Model-based Approach for Synthesis with Safe LTL

In the work of [43], the author translates safe LTL formulas to Deterministic Büchi Automata (DBA) [22] which are composed with a plant to form a specification which in turn can be synthesized to construct a supervisory controller. In the subsequent subsections we evaluate [43] based on the four guiding questions outlined in Section 7.3.

7.8.1 Specification Formalism Comparison

The author of [43] defines multiple constructs to model plants and specifications. In this section we will focus on Finite State Automaton (FSA) and safe LTL as they are most relevant to our evaluation. The author formally defines an FSA as a 4-tuple $\mathcal{A} = (Q, \Sigma, E, q_0)$ where:

- Q: The finite set of states.
- Σ: a finite set of events (the alphabet), partitioned into two disjoint sets: Σ_c for controllable events and Σ_u for uncontrollable events, i.e., $\Sigma := \Sigma_c \cup \Sigma_u$, $\Sigma_c \cap \Sigma_u = \emptyset$.
- $E : Q \times \Sigma \rightarrow Q$: The state transition relation.
- $q_0 \in Q$: The initial state.

There are notable differences between an FSA and an EFA in CIF (Definition 6.1). Firstly, there are no variables in an FSA, and consequently neither guards nor assignments on transitions. This can be mimicked, as described previously, by modeling each combination of variable valuations as separate states in the FSA. Secondly, there are no marked states in an FSA, where marked states are fundamental in CIF. This is a more fundamental limitation, which we will address in synthesis properties comparison section. Lastly, there is only a single initial state in an FSA, while CIF supports multiple initial states. Similar to CMG (Section 7.4), multiple initial states can be mimicked by introducing a new initial state and connecting it to the intended initial states using controllable transitions.

Additionally, requirements can be modeled using safe LTL formulas over the states of the FSA. Such formulas are translated into DBA which are then composed with the plant FSA to form a specification for synthesis. Safe LTL is a subset of LTL where the eventually operator F and until operator U are not allowed. The omission of these (liveness) operators ensures that all safe LTL formulas can be translated to DBA which are closed under union. Closed under union in this instance means that there are no strongly connected components composed solely of non-accepting states. Such components would allow for infinite runs that never reach an accepting state, violating the LTL property. Consider for example Figure 8 with strongly connected component S_0, accepting state S_1, and infinite run constructed by e_0^ω.

7.8.2 Synthesis Properties Comparison

The author's synthesis definitions align with classical RW-synthesis properties except for non-blockingness. Safety is defined such that the controlled system only allows sequences of events which adhere to the safe LTL specification and the plant. Controllability is defined such that the supervisor cannot disable uncontrollable events. Since there are no marked states in an FSA, non-blockingness is defined differently compared to classical supervisory control theory: Non-blockingness is defined such that the controlled system is deadlock-free, meaning there is always a possible transition from every reachable state. Maximal permissiveness is defined such that the minimal restrictions are imposed on the controlled system while ensuring safety, controllability, and deadlock-freeness.

7.8.3 Liveness Specification Capabilities

The author does not describe how liveness requirements can be specified, other than "something" must always be possible due to deadlock-freeness. Consequently, the liveness requirement of RWS, as defined in Section 7.3, cannot be specified using safe LTL.

7.8.4 Limitations

The most apparent limitation is the restriction to safe LTL formulas and the lack of liveness requirements in general. Nevertheless, the author provides a valuable model-based perspective on synthesis with LTL specifications.

7.9 Summary of Findings

This section summarizes the key findings from the covered literature on the topic of liveness specification in supervisor synthesis. Table 1 provides an overview of the various approaches discussed, highlighting the answers to the guiding questions outlined in Section 7.3. Maximally permissive is abbreviated as *max. permissive* in the table. In the RW-properties column, when a property is not listed, it means the approach does not guarantee that property.

From the table, several observations can be made regarding the state-of-the-art in liveness specification for supervisor synthesis. Firstly, it is clear that a variety of specification formalisms are employed across the different approaches, there is no universally adopted formalism for liveness specification in this context. This is probably due to the diverse requirements and constraints of different application domains.

When freedom in the controlled system is desired with guarantees on liveness in the form of reachability, approaches based on CTL or multitasking are suitable. However, full CTL is not (always) compatible with CIF and RW-synthesis in general due to scenarios when there are incomparable solutions, as discussed in Section 7.5.4. This breaks the uniqueness of maximal permissiveness as defined in classical RW-synthesis (Definition 6.24).

Compared to CTL, stronger liveness guarantees can be achieved using LTL-based approaches, in the form of infinitely often visiting desired states. But this comes at the cost of imposing significant restrictions on the possible behavior of the controlled system, as described in Section 7.6.4. Consequently, the LTL-based approaches, except for safe LTL [43], are not maximally permissive.

When looking at the compatibility with CIF, [13] and [33] seem most suitable. Both sources link their approach directly to the RW-framework [6] and mostly align, or can be adapted to align, with CIF's modeling constructs. With respect to applicability to the RWS case study (Section 5), similarly, these two approaches appear most promising. As described in Section 5, and more precisely in Section 7.3, RWS requires that multiple states remain reachable in the controlled system. This property can be expressed in the approach of [13] by coloring all desired states with a different color, and in the work of [33] by specifying multiple $AG\ EF\ \phi$ CTL requirements, where ϕ represents a desired state.

Lastly, we want to mention [43] as its interesting approach utilizes modeling constructs to express temporal logic properties. However, this approach faces limitations as it does not consider liveness requirements.

To conclude, while several approaches exist for specifying liveness in supervisor synthesis, each with its own strengths and weaknesses, the work of [13] and [33] clearly stand out as the most compatible with CIF and applicable to the RWS case study. However, both approaches are not fully aligned with CIF and consequently we refine the global research question described in the introduction (Section 3) to the following sub-research question: *How can the work of [13] and [33] be adopted in CIF?* While we partially answered this question in Subsections 7.4 and 7.5, a complete answer will be provided in Section 8. Furthermore, the work of [43] inspired us to explore the possibility of expressing the liveness requirement required by RWS using existing CIF constructs. This inspiration led to the following sub-research question: *If possible, how can the liveness requirement of RWS be expressed using existing CIF constructs?* We will likewise provide a complete answer to this question in Section 8.

Source	**Specification**	**RW Properties**	**Liveness Logic**	**RWS Requirement**	**Main Limitations**
[13]	CMG	Safety Controllability Non-blocking Max. permissive	Colors (multitasking)	Yes (via colors)	No variables Single initial state
[33]	LTS CIR (CTL subset)	Safety Controllability Non-blocking Max. permissive	$AG(\bigwedge_i p_i \wedge \bigwedge_j EF(p_j))$	Yes	No non-determinism Limited CTL support No initial state predicate
[35]	DES Canonical LTL	Safety Controllability Different non-blocking	$G(P \cdot \bigwedge_i F(M_i))$	No (stricter)	Very restricting Not max. permissive
[34]	TS LTL$_{\setminus 0}$	Safety Different non-blocking	LTL without next (X)	No (stricter)	Very restricting Not max. permissive Runtime overhead No events
[43]	FSA Safe LTL (no F, U)	Safety Controllability Max. permissive	Only deadlock-freeness	No (safety only)	No liveness require- ments

Table 1: Overview of covered approaches for liveness specifications in supervisor synthesis.

8 Proposed Approaches

In this section, we present two approaches based on the findings from the literature study in Section 7. First, we demonstrate how the approaches of [13] and [33] can be combined to extend the liveness capabilities of CIF as to solve the challenge faced by RWS (Sections 5 and 7.3). Secondly, we explore the possibility of expressing the liveness requirement of RWS using existing CIF constructs, leading to a novel approach which solely uses concepts of classical RW-synthesis. This second approach is inspired by the work of [43].

The first approach is detailed in Section 8.1, while the second approach is outlined in Section 8.2. For both approaches, we first provide an introduction followed by an explanation of the CIF code and the intuition behind it. Afterwards, we provide a formal definition of the approach. As we believe the second approach to be novel, we provide a proof of its correctness in Section 9.

8.1 Reachability Requirement Annotation

We combine the approaches of [13] and [33] by utilizing the algorithmic approach of [13] with the expressiveness of [33]. Every property that must remain reachable in the controlled system will be denoted as a *task*, such that *AG EF task* holds in the controlled system, if the controlled system is total (Definition 6.8). This CTL formula ensures that from every reachable state in the controlled system, there exists a path to a state satisfying the task, which is precisely the liveness requirement needed by RWS (as defined in Section 7.3). We slightly deviate from the semantics of CTL (Definition 6.31) in this approach as we do allow the controlled system to be non-total. The reason behind this decision is the fact that CIF already allows marked deadlocks (a marked state with no outgoing transitions) in the controlled system. When a guarantee on the totality of the controlled system is required, the approach outlined in Section 8.2 can be used instead.

8.1.1 CIF Code and Intuition

To specify multitasking in CIF, we introduce the `@requirement:reachable(task)` annotation. This annotation indicates that a state satisfying the given predicate (`task`) must remain reachable in the controlled system. The prefix "`@requirement:`" distinguishes it from other annotations and enables grouping it with possible future annotation-based requirements. Listing 3 shows an example of how RWS could use this annotation to specify that a generic lock gate must remain openable and closeable. Please note that this is a simplified example for illustrative purposes.

```
group GenericLock:
    group UpperHead:
        group DoorName1:
            @requirement:reachable(open)
            @requirement:reachable(closed)
            plant Actuator:
                location off:
                    initial; marked;
                    edge c_open goto open;
                    edge c_close goto closed;
                location open:
                    edge c_shutdown goto off;
                location closed:
                    edge c_shutdown goto off;
                ...
```

Listing 3: Simplified GenericLock highlighting multitasking syntax in CIF.

```
1  @@requirement:reachable(GenericLock.UpperHead.DoorName1.Actuator.
   open)
2
3  @requirement:reachable(UpperHead.DoorName1.Actuator.open)
4  group GenericLock:
5      @requirement:reachable(DoorName1.Actuator.open)
6      group UpperHead:
7          @requirement:reachable(Actuator.open)
8          group DoorName1:
9              @requirement:reachable(open)
10             plant Actuator:
11                 ...
```

Listing 4: Reachability annotation expressing the same requirement.

The annotation can be attached to various CIF constructs to enable references without the need to provide the full namespace. Listing 4 illustrates this aspect as it shows how the same reachability requirement can be expressed in five different ways. The first annotation (line 1) is attached to the specification itself by placing it at the top of the file and prefixing it with an additional "@". The subsequent four annotations (line 3, 5, 7, and 9) highlight how the predicate name length shrinks the closer the annotation is placed to the construct it refers to.

The syntax of annotations differs from the original color markings used in [13]. In their approach, individual locations are marked with colors to indicate which locations must remain reachable. Since CIF specifications can contain variables and complex predicates over locations, we opted for a more flexible annotation approach using predicates. This change in syntax was inspired by the expressiveness of the liveness requirements in [33].

8.1.2 Formal Definition

In the work of [33], we have seen that EFA-like constructs can be transformed to FAs by flattening all possible variable valuations. This is likewise observable in the definition of an EFA (Definition 6.1) for which its full state-space is given by the product of its locations and possible variable valuations. We use this property to employ the algorithm of [13]. More specifically, we use the work of [46], which provides a clear description of how to add multitasking to RW-synthesis.

Multitasking can be added to CIF's synthesis loop (Section 6.3.3) by introducing an additional backwards reachability search per task, as shown in lines 5-7 of Algorithm 3. These additional searches ensure that all tasks remain reachable in the controlled system by strengthening initial state predicates or guards on transitions to states for which this property is violated. Similar to how safety and non-blocking requirements are enforced, line 8 now additionally ensures that no states are reachable in the controlled system that lead uncontrollably to a state from which a task can no longer be fulfilled. In other words, states are removed if an uncontrollable event can lead from that state to another state from which no path exists to any state satisfying the task. This ensures controllability is maintained while guaranteeing task reachability.

By adding reachability annotations to CIF and adapting the synthesis algorithm accordingly, we enable the specification and synthesis of supervisors that ensure multiple liveness properties are satisfied in the controlled system. This provides RWS with the necessary liveness guarantees to solve their challenge (as described in Sections 5 and 7.3) while maintaining compatibility with CIF's modeling constructs and the RW-framework. The reachability annotation has been added in CIF version 10.0.

Algorithm 3 Multitasking Symbolic Supervisor Synthesis (*MSSS*), originally from [9] with adaptations for multitasking from [46].

```
Input: Specification SEFA $M = (V, D, \Sigma, E, p_0, p_m)$, forbidden states $p_f$ over $V$, and
    tasks $T$ over $V$.
Output: Controlled-system SEFA $S$.
 1: $C \leftarrow \neg p_f$
 2: repeat
 3:     $C' \leftarrow C$
 4:     $C \leftarrow BRS(p_m, E, C)$
 5:     for all $t \in T$ do                    ▷ Additional loop to enable multitasking.
 6:         $C \leftarrow BRS(t, E, C)$
 7:     end for
 8:     $B \leftarrow BRS(\neg C, E_u, true)$
 9:     $C \leftarrow \neg B$
10:     $C \leftarrow FRS(p_0, E, C)$                              ▷ Optional step.
11: until $C = C'$
12: for all $e \in E$ with $\sigma \in \Sigma_c$ do
13:     $g \leftarrow g \wedge relprev(e, C)$
14: end for
15: $S \leftarrow (V, D, \Sigma, E, p_0 \wedge C, p_m \wedge C)$
```

8.2 Reachability Requirement Automaton

Inspired by the work of [43], we explored expressing the liveness requirement needed by RWS using existing CIF constructs. This led to what we believe to be a novel and elegant approach that relies solely on classical RW-synthesis concepts. The CIF code for this approach is given in Listing 5 and its EFA representation in Figure 9. We first cover the CIF code and intuition in Section 8.2.1, after which we formally define the approach in Section 8.2.2. Subsequently, we provide a correctness proof in Section 9.

8.2.1 CIF Code and Intuition

Line 1 of the CIF model displayed in Listing 5 uses a requirement automaton definition (Section 4.3) to enable multiple instantiations of the automaton with different predicates ϕ (`phi` in the CIF model). In line 2, the uncontrollable event `t` is defined, which is the only event used in the automaton. Afterwards, in lines 3-8, two locations q_g and q_{ng} (`q_g` and `q_ng` respectively in the CIF model) are defined. q_g is the initial and marked location which includes a t-transition to q_{ng} that is always enabled (line 5). On the other hand, q_{ng} is neither initial nor marked and contains two t-transitions: one defined in line 7 with guard ϕ leading back to q_g, and another in line 8 as self-loop with guard $\neg\phi$ to make sure there is always an enabled t-transition.

The core idea is to instantiate a small requirement automaton for each predicate ϕ that must remain satisfiable in the controlled system, effectively enforcing $AG\ EF\ \phi$. This automaton uses controllability (Definition 6.20) and non-blockingness (Definition 6.21) to ensure that states from which ϕ is not reachable can transition uncontrollably to blocking states. Because uncontrollable transitions cannot be disabled (Definition 6.20), these blocking states must be removed by strengthening guards of previous controllable transitions or by restricting initial state predicates.

This approach can be used by RWS to ensure a waterway lock gate is both openable and closable by including the requirement definition of Listing 5 and instantiating it twice, as shown in Listing 6. Note that, similar to the reachable annotation displayed in Listing 4, this is a simplified example for illustrative purposes.

In this subsection, we provided an intuitive explanation of the CIF code for the reachability requirement automaton. We defined an automaton definition which can be used to instantiate requirement automata that leverage classical RW concepts (controllability and non-blockingness) to enforce the liveness requirement needed by RWS. Next, we provide a formal definition of the approach, which we use in the correctness proof in Section 9.

```
1 requirement def AGEF(alg bool phi):
2     uncontrollable t;
3     location q_g:
4         initial; marked;
5         edge t goto q_ng;
6     location q_ng:
7         edge t when phi goto q_g;
8         edge t when not phi;
9 end
```

Listing 5: Reachability requirement automaton in CIF.

```
R1 : AGEF(GenericLock.Upperhead.Doornam1.Actuator.open);
R2 : AGEF(GenericLock.Upperhead.Doornam1.Actuator.closed);

group GenericLock:
    group UpperHead:
        group DoorName1:
            plant Actuator:
                location off:
                    initial; marked;
                    edge c_open goto open;
                    edge c_close goto closed;
                location open:
                    edge c_shutdown goto off;
                location closed:
                    edge c_shutdown goto off;
                ...
```

Listing 6: Instantiations of the reachability requirement automaton (Listing 5) for RWS.

8.2.2 Formal Definition and Supporting Properties

In this subsection, we begin by formally defining the function $R_{AGEF\phi}$, which we introduced in the previous subsection as a requirement automaton definition. We will denote automata produced by this function as R_ϕ. Afterwards, we describe plantification and parallel composition of EFAs produced by $R_{AGEF\phi}$. Subsequently, we establish a simulation relation between an arbitrary specification and that specification composed (Definition 6.10) with R_ϕ. We will use the formal description and properties established in this section as building blocks for the correctness proof in Section 9. Formally, $R_{AGEF\phi}$ is defined as a function producing EFAs (Definition 6.1) as follows:

Definition 8.1 ($R_{AGEF\phi}$)**.** Consider a function $R_{AGEF\phi}$ that takes a predicate ϕ and a specification EFA M as input and outputs a requirement EFA defined as:

$$\begin{aligned}
R_{AGEF\phi}(\phi, M) := (Q, V, D, \Sigma, E, p_0, p_m) \text{ where:}& \\
Q &:= \{\, q_g,\, q_{ng} \,\}. \\
D &:= D_M \quad \text{(no variables introduced, shared with the specification)}. \\
V &:= V_M \quad \text{(no variables introduced, shared with the specification)}. \\
\Sigma &:= \Sigma_u \cup \Sigma_c. \\
\Sigma_u &:= \{\, t \,\} \text{ where } t \notin \Sigma_M. \\
\Sigma_c &:= \emptyset. \\
E &:= \{ \\
&\quad (q_g, t, \texttt{true}, \texttt{false}, q_{ng}), \\
&\quad (q_{ng}, t, \phi, \texttt{false}, q_g), \\
&\quad (q_{ng}, t, \neg\phi, \texttt{false}, q_{ng}) \\
&\}. \\
p_0 &:= \{\, q_g \,\}. \\
p_m &:= \{\, q_g \,\}.
\end{aligned}$$

Furthermore, we forbid any specification M from referencing locations q_g and q_{ng}, as this could interfere with the intent of the requirement.

Figure 9 shows a graphical representation of R_ϕ. In this figure, t is the uncontrollable transition unique to R_ϕ. This figure highlights the simplicity of R_ϕ, which only consists of two states and three transitions. Nevertheless, as we will show in Section 9, it effectively influences synthesis to enforce $AG\,EF\,\phi$ in the controlled system. But first, we will establish two supporting properties that will be used in the correctness proof.

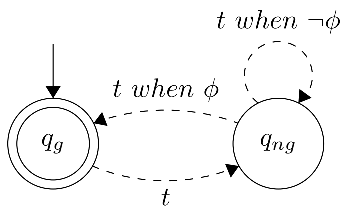


Figure 9: Graphical representation of EFAs resulting from $R_{AGEF\phi}$.

Plantification Since the output R_ϕ of $R_{AGEF\phi}$ is a requirement automaton, it will be plantified before being composed with other plants and requirements (as described in Section 6.3.1). We will establish in the following proposition that this process will not introduce any state/event exclusion invariants. This property is important in establishing safe language equivalence outlined in Section 9.1.

Proposition 8.2 (Plantifying outputs of $R_{AGEF\phi}$ does not result in state/event exclusion invariants)**.** *Consider R_ϕ as the result of $R_{AGEF\phi}$ given a predicate ϕ and specification M. Then, plantification of R_ϕ will not introduce any state/event exclusion invariants.*

Proof. In both locations of R_ϕ, q_g and q_{ng}, there is always an edge where t is enabled. For q_g, t is enabled when the current location is q_g since the guard is simply *true*. For q_{ng}, t is enabled when the current location is q_{ng} and $\phi \lor \neg\phi$ holds, which is always true. Since both guards always evaluate to *true*, the combined guard evaluates to *true*. Therefore, no extra self-loop edges are generated during plantification and thus no state/event exclusion requirement invariant is generated for R_ϕ, making the relabeling from requirement to plant the only difference after plantification. □

Parallel Composition After plantification, R_ϕ can be composed in parallel (Definition 6.10) with other EFAs. The parallel composition M' of EFA M and $R_\phi := R_{AGEF\phi}(\phi, M)$, given a predicate ϕ, is then defined as $M' := M||R_\phi$, following the rules of parallel composition (Definition 6.10). The components of M' are derived as follows:

- $Q_{M'} := Q_M \times Q_{R_\phi} = Q_M \times \{q_g, q_{ng}\}$.
- $V_{M'} = V_M$.
- $D_{M'} = D_M$.
- $\Sigma_{M'} = \Sigma_M \cup \Sigma_{R_\phi} = \Sigma_M \cup \{t\}$, with $t \notin \Sigma_M$.
- $p_{0,M'} = p_{0,M} \land p_{0,R_\phi}$.
- $p_{m,M'} = p_{m,M} \land p_{m,R_\phi}$.
- $E_{M'}$: As the alphabets of M and R_ϕ are disjoint ($\Sigma_M \cap \Sigma_{R_\phi} = \emptyset$), the transition relation $E_{M'}$ combines transitions unique to M and transitions unique to R_ϕ as follows:
  - For events $\sigma \in \Sigma_M$: If M has a transition $(q_M, g_M, r_M, \sigma, u_M, q'_M)$, then M' will have a transition $((q_M, q_{R_\phi}), g_M, r_M, \sigma, u_M, (q'_M, q_{R_\phi}))$ for each $q_{R_\phi} \in Q_{R_\phi}$. In this case, the state of R_ϕ remains unchanged.

– For the event $t \in \Sigma_{R_\phi}$: If R_ϕ has a transition $(q_{R_\phi}, g_{R_\phi}, r_{R_\phi}, t, u_{R_\phi}, q'_{R_\phi})$, then M' will have a transition $((q_M, q_{R_\phi}), g_{R_\phi}, r_{R_\phi}, t, u_{R_\phi}, (q_M, q'_{R_\phi}))$ for each $q_M \in Q_M$. Here, the state of M remains unchanged.

Simulation Relation To demonstrate that M' preserves the language, states, and predicate evaluation of M, we establish a simulation relation (Definition 6.16) between the two EFAs.

Proposition 8.3 (M is simulated by M')**.** *Consider an EFA M, a predicate ϕ, $R_\phi := R_{AGEF\phi}(\phi, M)$, and $M' := M \| R_\phi$. Then M is simulated by M', i.e., $M \preceq M'$. Consequently, the language of M is equivalent to that of M' after projection: $L(M) = Proj_{\Sigma_M}(L(M'))$.*

Proof. We construct a simulation relation (Definition 6.16) Sim $\subseteq X_M \times X_{M'}$ as follows: for every state $x_M \in X_M$, we include $(x_M, (x_M, x_{R_\phi})) \in$ Sim.

1. **Initial states**: For any initial state $x_{0,M} \in X_M^0$, the corresponding initial state in M' is $(x_{0,M}, q_g)$ (since $q_g \iff p_{0,R_\phi}$). Thus, the initial condition for simulation holds.

2. **Predicate satisfaction**: For any state x_M of M and any predicate p satisfied in x_M ($x_M \models p$), there exists a corresponding state $(x_M, x_{R_\phi}) \in X_{M'}$, such that $(x_M, x_{R_\phi}) \models p$. By simulation (Definition 6.16), p must be a predicate over X_M and is therefore not influenced by x_{R_ϕ}, hence $x_M \models p \implies (x_M, x_{R_\phi}) \models p$.

3. **Transition preservation**: For any transition $x_M \xrightarrow{\sigma} x'_M$ possible in M with $\sigma \in \Sigma_M$:

   - Since $\sigma \in \Sigma_M$, σ is necessarily not in Σ_{R_ϕ} (Definition 8.1). Then, as described in the parallel composition of R_ϕ 8.2.2, if $x_M \xrightarrow{\sigma} x'_M$ is possible in M, there exists a possible transition $(x_M, x_{R_\phi}) \xrightarrow{\sigma} (x'_M, x_{R_\phi})$ in M'.
   - The guard, error predicate, and update function are identical to those in M, ensuring the same transitions are enabled.
   - Thus, if $(x_M, (x_M, x_{R_\phi})) \in$ Sim, then there exists a transition $(x_M, x_{R_\phi}) \xrightarrow{\sigma} (x'_M, x_{R_\phi})$ in M', and by our relation, $(x'_M, (x'_M, x_{R_\phi})) \in$ Sim.

4. t **does not modify** M: The event $t \in \Sigma_{R_\phi}$ is not included in Σ_M. Transitions involving t in M' only modify the R_ϕ component and interleave with transitions in M. Therefore, from the perspective of Σ_M, the event t is invisible to the simulation relation.

This construction demonstrates that $M \preceq M'$. Consequently, every word $w \in L(M)$ is also possible in M'. Additionally, any word in $L(M')$ projected onto Σ_M corresponds to a run in $runs(M, L(M))$ (since R_ϕ does not impose restrictions on M and only transitions within the M component change its state). Thus: $L(M) = Proj_{\Sigma_M}(L(M'))$. □

In this subsection, we provided a formal definition of the reachability requirement automaton generating function $R_{AGEF\phi}$ and established two supporting properties. We will now use these properties to prove the correctness of the approach in the following section.

9 $R_{AGEF\phi}$ Correctness Proof

The goal of this section is to prove that $R_{AGEF\phi}$ correctly produces a requirement automaton R_ϕ which, when composed with the specification, enforces $AG\,EF\,\phi$ to hold in all initial states of the synthesized controlled system.

Consider a specification M, an arbitrary predicate ϕ, a requirement automaton $R_\phi := R_{AGEF\phi}(\phi, M)$, a combined specification $M' := M \| R_\phi$, the forbidden state predicate p_f for M', a supervisor $S' := \mathit{Synth}(\mathit{ToSEFA}(M'), p_f)$, and a controlled system $C' := M' \| S'$. We will prove that $KS(C', L(C')) \models AG\,EF\,\phi$ holds (Theorem 9.15) in a maximally permissive manner. Recall that by definition the definition of CTL (Definition 6.31), $KS(C', L(C')) \models AG\,EF\,\phi$ holds iff $AG\,EF\,\phi$ holds in all initial states of C'.

This correctness proof is structured according to the three fundamental properties that supervisors must satisfy: safety, controllability, and non-blockingness. We analyze how R_ϕ influences each property and demonstrate that these influences collectively enforce $AG\,EF\,\phi$ in a minimally restrictive manner. Maximal permissiveness follows as a consequence of these properties.

We will now provide an overview of how R_ϕ influences each property before presenting the formal proofs in the following subsections. With respect to **safety** (Section 9.1), R_ϕ does not contribute to the forbidden state predicate p_f and simulates M, so therefore the safe language of M is preserved after projecting away t (Theorem 9.3). Hence, no additional safety restrictions are imposed by R_ϕ. Regarding **controllability** (Section 9.2), R_ϕ does not impose any direct restrictions on the uncontrollable transitions of M (Theorem 9.4). It only introduces its own unique uncontrollable event $t \in \Sigma_{u,R_\phi}$, which is always enabled. For **non-blockingness** (Section 9.3), R_ϕ includes an unmarked location q_{ng} that is always reachable via the uncontrollable transition t. To satisfy non-blockingness, synthesis must ensure a path back to a marked location exists from every reachable state. In R_ϕ, the only path from q_{ng} to the marked location q_g requires ϕ to hold. This forces synthesis to restrict the specification such that from every reachable state, a state satisfying ϕ remains reachable, thereby enforcing $AG\,EF\,\phi$. Finally, **maximal permissiveness** (Section 9.4) is preserved because R_ϕ only influences non-blockingness by imposing the minimal restrictions necessary to satisfy $AG\,EF\,\phi$, without affecting safety or directly influencing controllability.

9.1 Safety

We will show that the addition of R_ϕ fully maintains the safe language after projecting away t. The safe language of an EFA M is determined by three factors: the language $L(M)$, the reachable states $\mathit{States}(\mathit{Runs}(M, L(M)))$, and the forbidden state predicate p_f, as can be derived from the safe language definition (Definition 6.19): $L_s(M) := \{\, w \in L(M) \mid \forall x \in \mathit{States}\Big(\ \mathit{Runs}(M, \{w\})\ \Big) : \ x \models \neg p_f \,\}$. M' and M will be compared on all three aspects in the following sections where we conclude in Theorem 9.3 that the safe language is preserved after projection: $L_s(M) = \mathit{Proj}_{\Sigma_M}(L_s(M'))$.

9.1.1 Language Preservation

From the simulation relation $M \preceq M'$, we have already established that $L(M) = \mathit{Proj}_{\Sigma_M}(L(M'))$, in 8.3. This ensures that any word possible in M (with respect to Σ_M) is also possible in M', and any projected word from M' corresponds to a possible word in M.

9.1.2 Reachable State-Space Preservation

The parallel composition $M' := M \| R_\phi$ constructs the state-space as $X_{M'} = X_M \times X_{R_\phi}$. For every state $x_M \in X_M$ reachable in M, there exists a corresponding state $(x_M, q_{R_\phi}) \in X_{M'}$ reachable in M' for some $q_{R_\phi} \in \{q_g, q_{ng}\}$. The simulation relation $M \preceq M'$ (Proposition 8.3) guarantees that every run in $\mathit{Runs}(M, L(M))$ has a corresponding run in $\mathit{Runs}(M', L(M))$, which means that $\mathit{States}\Big(\mathit{Runs}(M, L(M))\Big) = \Big\{x_M \in X_M \mid \exists q_{R_\phi} \in Q_{R_\phi} : (x_M, q_{R_\phi}) \in \mathit{States}\Big(\mathit{Runs}(M', L(M'))\Big)\Big\}$.

Additionally, since the event t only changes the R_ϕ component and does not affect the M component, any reachable state $(x_M, q_{R_\phi}) \in X_{M'}$ corresponds to a state x_M that would be reachable in M through a word of $L(M)$. Therefore, all reachable states of M can be reached by $\mathit{Proj}_{\Sigma_M}(L(M'))$, as $L(M) = \mathit{Proj}_{\Sigma_M}(L(M'))$ by 9.1.1.

9.1.3 Forbidden State Predicate Preservation

Lemma 9.1 (R_ϕ solely introduces a location pointer variable)**.** *Consider a specification EFA M, a predicate ϕ over the state of M, and a requirement automaton $R_\phi := R_{AGEF\phi}(\phi, M)$. Then, during the transformation from an EFA to a SEFA, R_ϕ only introduces a variable implicitly, in the form of a location pointer.*

Proof. By Definition 8.1, R_ϕ is constructed without adding explicit variables. Definition 6.13 shows the EFA to SEFA transformation represents state through location pointers, and no other variables are introduced. Thus, after R_ϕ is transformed to a SEFA, only the location pointer is introduced. □

Proposition 9.2 (Forbidden state predicate preservation)**.** *The forbidden state predicate of M' always evaluates to the same truth value as that of M, i.e., $p_{f_{M'}} \equiv p_{f,M}$.*

Proof. The computation of a forbidden state predicate p_f consists of four steps, as described in section 6.3.2. $p_{f_{M'}}$ is logically equivalent to p_{f_M} in all four steps:

1. p_r is computed by the conjunction of all requirement invariants. $p_{r,M'} \equiv p_{r,M}$, because the addition of R_ϕ to M does not introduce invariants, as proven in 8.2.

2. p_f is initialized as $\neg p_r$, since $p_{r,M'} \equiv p_{r,M}$ implies $p_{f,M'} \equiv p_{f,M}$, in this step.

3. p_f is updated with a predicate representing states for which requirements' variables go out of range. R_ϕ only introduces a location pointer variable (implicitly), by lemma 9.1. However, by construction of location pointer variables during linearization 6.12, they can never be updated outside their range. Implying, $p_{f,M'} \equiv p_{f,M}$, in this step.

4. In the final step, states from which there exists an uncontrollable transition to a forbidden state are also considered forbidden and added to p_f. By Definition 8.1, the states of R_ϕ are not forbidden, as they cannot be referenced by other EFAs in the specification and are also not considered forbidden by R_ϕ itself. Furthermore, the uncontrollable transitions of R_ϕ are also not forbidden, as they are always enabled (Proposition 8.2). Concluding, $p_{f,M'} \equiv p_{f,M}$ also holds in this final step.

□

9.1.4 Safe Language Equality After Projection

Theorem 9.3 (Safe language preservation). *Consider a specification EFA M, a predicate ϕ over the state of M, a requirement automaton $R_\phi := R_{AGEF\phi}(\phi, M)$, and a combined specification $M' := M \| R_\phi$. Then, the safe language of M is preserved under the parallel composition with R_ϕ, after projection onto Σ_M, i.e., $Proj_{\Sigma_M}(L_s(M')) = L_s(M)$.*

Proof. By Definition 6.19, the safe language of M is given by:
$L_s(M) = \big\{ w \in L(M) \mid x \in \text{States}\big(\text{Runs}(M, \{w\})\big) : x \models \neg p_{f,M} \big\}$.

From language preservation (Section 9.1.1), $L(M) = Proj_{\Sigma_M}(L(M'))$, it follows that:
$L_s(M) = \big\{ w \in Proj_{\Sigma_M}(L(M')) \mid x \in \text{States}\big(\text{Runs}(M, \{w\})\big) : x \models \neg p_{f,M} \big\}$.

From the simulation relation $M \preceq M'$ (Propostion 8.3) follows reachable state preservation and state predicate preservation, therefore:
$L_s(M) = \big\{ w \in Proj_{\Sigma_M}(L(M')) \mid x \in \text{States}\big(\text{Runs}(M', \{w\})\big) : x \models \neg p_{f,M} \big\}$.

From forbidden state predicate preservation (Proposition 9.2), where $p_{f,M} \equiv p_{f,M'}$, it follows that:
$L_s(M) = \big\{ w \in Proj_{\Sigma_M}(L(M')) \mid x \in \text{States}\big(\text{Runs}(M', \{w\})\big) : x \models \neg p_{f,M'} \big\}$.

We will now show that the right hand side is equivalent to $Proj_{\Sigma_M}(L_s(M'))$.

By Definition 6.19, the safe language of M' is given by:
$L_s(M') = \big\{ w \in L(M') \mid x \in \text{States}\big(\text{Runs}(M', \{w\})\big) : x \models \neg p_{f,M'} \big\}$.

Consider a word $w \cdot t \cdot v$ in $L_s(M')$. By Definition 6.19, the forbidden state predicate $p_{f,M'}$ should not hold for all states along the run:
$\forall x \in \text{States}\big(\text{Runs}(M', \{w \cdot t \cdot v\})\big) : x \models \neg p_{f,M'}$.

Since t does not influence the truth value of $p_{f,M'}$ (Proposition 9.2), the states along the run of $w \cdot v$ also satisfy $\neg p_{f,M'}$:
$\forall x \in \text{States}\big(\text{Runs}(M', \{w \cdot v\})\big) : x \models \neg p_{f,M'}$.

As t is always enabled in M' (Proposition 8.2) and it interleaves with all other events (by Definition 8.1), we can omit it from a word in the language and still have a valid word of the language. That is, if $w \cdot t \cdot v \in L(M')$, then $w \cdot v \in L(M')$. By Definition 6.19, $L_s(M')$ is constructed from all possible words $L(M')$, by filtering out words that lead to forbidden states. Therefore, $w \cdot v$ also belongs to $L_s(M')$, i.e., $\forall w \cdot t \cdot v \in L_s(M') : w \cdot v \in L_s(M')$.

This means that we can project away t from the language of M' in the construction of $L_s(M')$ and get $Proj_{\Sigma_M}(L_s(M'))$ (recall that $\Sigma_M = \Sigma_{M'} \setminus \{t\}$):
$Proj_{\Sigma_M}(L_s(M')) = \big\{ w \in Proj_{\Sigma_M}(L(M')) \mid x \in \text{States}\big(\text{Runs}(M', \{w\})\big) : x \models \neg p_{f,M'} \big\}$.

Since we have proven that the right hand side is equivalent to $L_s(M)$, we conclude:
$Proj_{\Sigma_M}(L_s(M')) = L_s(M)$. □

9.2 Controllability

We now show that the addition of $R_\phi := R_{AGEF\phi}(\phi, M)$ to EFA specification M through parallel composition does not impose any new restrictions on the uncontrollable transitions of M. Intuitively, R_ϕ acts as a passive observer: its own uncontrollable event t is always enabled (as described in the plantification, Definition 8.2.2), and it interleaves with events of M since their alphabets are disjoint. Hence, the controllability properties of M are fully preserved in $M' := M \| R_\phi$.

Theorem 9.4 (Controllability preservation)**.** *Consider a specification EFA M, a predicate ϕ over the state of M, a requirement automaton $R_\phi := R_{AGEF\phi}(\phi, M)$, and a combined specification $M' := M \| R_\phi$. Then R_ϕ does not directly restrict any uncontrollable transitions.*

Proof. This property follows from four established properties:

1. The simulation relation $M \leq M'$ (Proposition 8.3).
2. The disjoint alphabets, $\Sigma_{R_\phi} \cap \Sigma_M = \emptyset$ (by construction, Definition 8.2.2).
3. Safe language preservation (Theorem 9.3).
4. t is always enabled (by Proposition 8.2).

From the simulation relation, every transition in M (including uncontrollable transitions) has a matching transition in M' over the same event. Because the alphabets are disjoint, the parallel composition does not require any synchronization on $\sigma \in \Sigma_M$. Hence, each such event remains enabled in M' exactly as it is in M. In other words, R_ϕ does not disable any uncontrollable transitions of M.

Finally, as described in Proposition 8.2, the uncontrollable event $t \in \Sigma_{R_\phi}$ itself is always enabled. Concluding, R_ϕ does not impose any direct restrictions on uncontrollable events. □

9.3 Non-blockingness

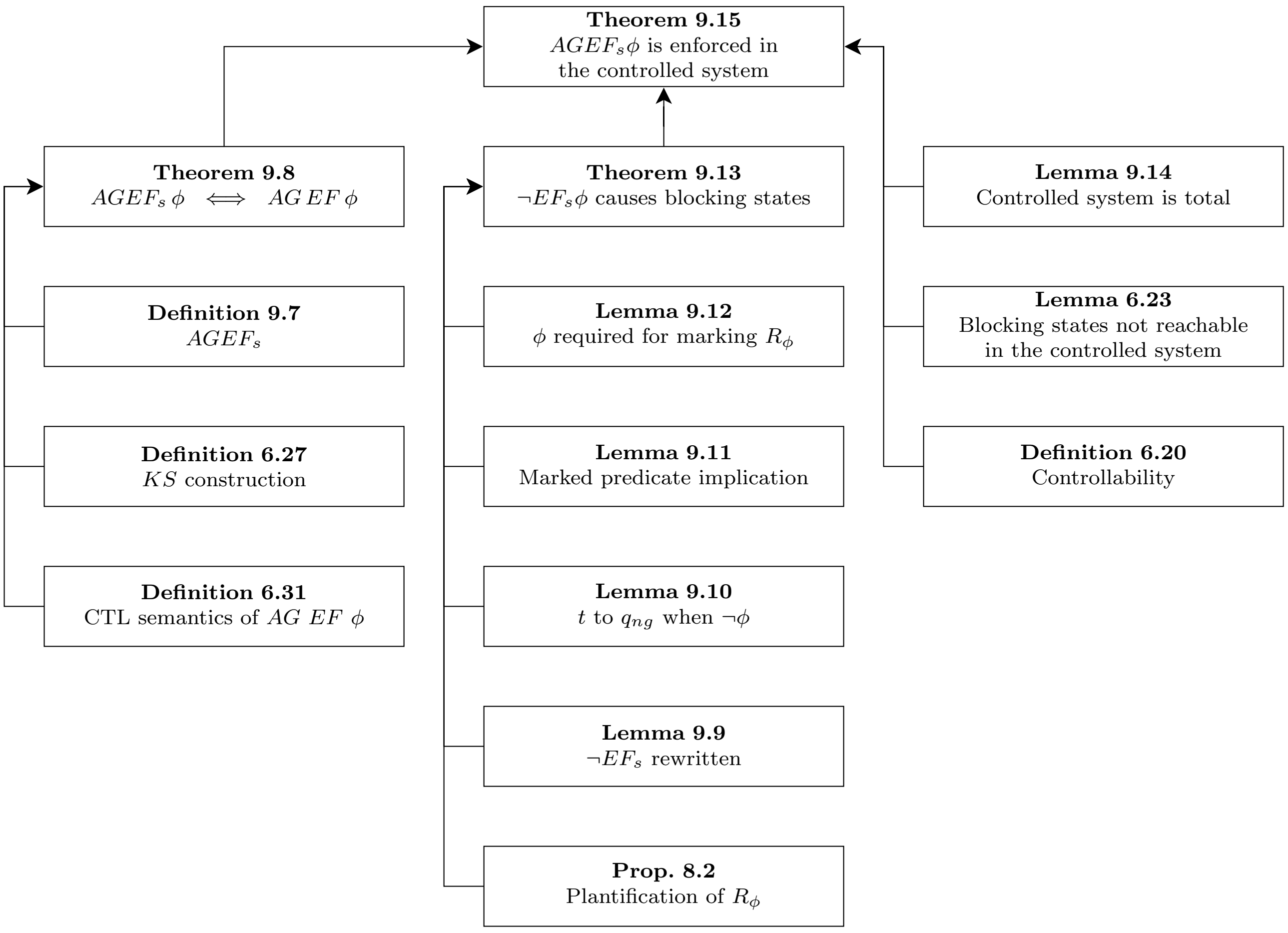
In this section, we prove how the requirement automaton R_ϕ (produced by $R_{AGEF\phi}$, Definition 8.1, for a specification M and predicate ϕ) enforces the liveness requirement $AGEF(\phi)$. We will denote a specification composed with R_ϕ as M', such that $M' := M \| R_\phi$. Furthermore, we will refer to the SEFA variants of M and M' as M^s and M'^s respectively.

R_ϕ is constructed such that from its unmarked state there is only a path to its marked state when ϕ holds. This utilizes the non-blocking property of synthesis (Definition 6.21) which states that a path to a marked state in the controlled system must always exist. Furthermore, R_ϕ only contains uncontrollable events, and consequently, by Definition 6.20, its restrictions must be propagated back to controllable transitions or the initial state predicate. Both aspects combined, influence synthesis in such a manner as to enforce $AG\ EF\ \phi$ for all initial states in the controlled system.

We divide this proof into three sections, one for each of the theorems:

1. **SEFA Equivalent CTL Functions (Section 9.3.1).**
   In this section we formally link our (SEFA) approach to CTL semantics.
2. **Unsatisfiability of ϕ Causes Blocking States (Section 9.3.2).**
   In this section we establish that the addition of R_ϕ to a specification results in uncontrollable transitions to blocking states when ϕ is unsatisfiable.
3. $AG\ EF\,\phi$ **is Enforced in the Controlled System (Section 9.3.3).**
   In this section we combine the proofs of the preceding sections to establish that $AG\ EF\ \phi$ is enforced in all reachable states of the controlled system.

Figure 10 highlights the most important dependencies of Theorems 9.8, 9.13, and 9.15. Please note that this is not a complete dependency graph, as some lemmas and definitions have been omitted for clarity.



Theorem 9.15
$AGEF_s\phi$ is enforced in the controlled system
Theorem 9.8
$AGEF_s\,\phi \iff AG\,EF\,\phi$
Theorem 9.13
$\neg EF_s\phi$ causes blocking states
Lemma 9.14
Controlled system is total
Definition 9.7
$AGEF_s$
Lemma 9.12
ϕ required for marking R_ϕ
Lemma 6.23
Blocking states not reachable in the controlled system
Definition 6.27
KS construction
Lemma 9.11
Marked predicate implication
Definition 6.20
Controllability
Definition 6.31
CTL semantics of $AG\ EF\ \phi$
Lemma 9.10
t to q_{ng} when $\neg\phi$
Lemma 9.9
$\neg EF_s$ rewritten
Prop. 8.2
Plantification of R_ϕ

Figure 10: Graph displaying the most important dependencies of the main theorems relating to non-blockingness and $AG\ EF\ \phi$.

9.3.1 SEFA Equivalent CTL Functions

In this subsection we establish a SEFA equivalent function for the CTL formula $AG\,EF\,\phi$. This equivalence is crucial in linking our approach to the well established computational tree logic, providing a theoretical foundation for our method. We begin by defining the SEFA equivalents for the CTL operators AG and EF, in Definitions 9.5 and 9.6 respectively. Afterwards, we compose these definitions to form $AGEF_s$ in Definition 9.7 and prove it is semantically equivalent to $AG\,EF\,\phi$ in Theorem 9.8.

Definition 9.5 (Always)**.** Consider a function AG_s which takes as input a SEFA M, where M is total (Definition 6.8), a state $x \in X$, and a proposition ϕ and outputs *true* iff all reachable states of x satisfy ϕ. Formally, we define AG_s as follows:

$$AG_s(M, x, \phi) := \forall w \in \Sigma_M^*, \forall x_\phi \in X_M : \Big((x \xrightarrow{w} x_\phi) \implies (x_\phi \models \phi)\Big)$$

Definition 9.6 (Reachable)**.** Consider a function EF_s which takes as input a SEFA M, where M is total (Definition 6.8), a state $x \in X$, and a proposition ϕ and outputs *true* iff there exists a path to a state where ϕ holds. Formally, we define EF_s as follows:

$$EF_s(M, x, \phi) := \exists w \in \Sigma_M^*, \exists x_\phi \in X_M : \Big((x \xrightarrow{w} x_\phi) \wedge (x_\phi \models \phi)\Big)$$

Having established AG_s and EF_s, we can define their composition.

Definition 9.7 (Always reachable)**.** Consider a function $AGEF_s$ which takes as input a SEFA M, where M is total (Definition 6.8), a state $x \in X$, and a proposition ϕ and outputs *true* iff for all reachable states x' from x there exists a path to a state where ϕ holds. Formally, we define $AGEF_s$ as follows:

$$AGEF_s(M, x, \phi) := AG_s(M, x, \lambda x'.EF_s(M, x', \phi))$$

The semantic correctness of this composition is proven in the following theorem.

Theorem 9.8 (Equivalence to $AG\,EF\,\phi$)**.** *Consider a specification SEFA M^s, where M^s is total (Definition 6.8), a predicate ϕ over the state of M, and a Kripke structure $K := KS(M, L(M))$. Then, the function $AGEF_s(M^s, x, \phi)$ holds iff the CTL formula $AG\,EF\,\phi$ holds in the corresponding state of K. Formally:*
$AGEF_s(M^s, x, \phi) \iff \Big((K, x) \models AG\,EF\,\phi\Big)$

Proof. We will prove both directions of the iff-implication.
Direction 1: $AGEF_s(M^s, x, \phi) \implies \Big((K,\ x) \models AG\,EF\,\phi\Big)$
Assume $AGEF_s(M^s, x, \phi)$ holds, as otherwise the implication is trivially satisfied. By Definition 9.7, this means:
$AG_s(M, x, EF_s(M, x, \phi))$

From the definitions of AG_s and EF_s, Definition 9.5 and Definition 9.6 respectively, follows:
$\forall w \in \Sigma^*_{M^s}, \forall x' \in X_{M^s}:$
$\Big((x \xrightarrow{w} x') \implies \big(\exists v \in \Sigma^*_{M^s}, \exists x_\phi \in X_{M^s} : (x' \xrightarrow{v} x_\phi) \wedge (x_\phi \models \phi)\big)\Big)$

By CTL semantics (Definition 6.31), $(K, x) \models AG\,EF\,\phi$ means:
$\forall r \in Runs_{ks}(K, x), \forall i \geq 0 : \exists r' \in Runs_{ks}(K, r[i]), \exists j \geq 0 : r'[j] \models \phi$

As we assume M^s is total, all of its runs can be extended indefinitely and thus infinite runs are possible in K. Consider any infinite run $r = x_0 \to x_1 \to \cdots$ starting from $x_0 = x$ in S_K, and any state x_i, where $i \geq 0$. Since $AGEF_s(M^s, x, \phi)$ holds, there exists $v \in \Sigma^*_{M^s}$ and $x_\phi \in X_{M^s}$ such that:
$x_i \xrightarrow{v} x_\phi \wedge (x_\phi \models \phi)$

This path from x_i to x_ϕ in M^s corresponds (by Definition 6.27) to a path in K. Therefore, there exists a run $r' \in Runs_{ks}(K, r[i])$ and a $j \geq 0$ such that $r'[j] = x_\phi$ and $r'[j] \models \phi$. Since this holds for any run r and any reachable state x_i from x, we derive:
$(K, x) \models AG\,EF\,\phi$

Thus we conclude:
$AGEF_s(M^s, x, \phi) \implies \Big((K,\ x) \models AG\,EF\,\phi\Big)$

Direction 2: $\Big((K,\ x) \models AG\,EF\,\phi\Big) \implies AGEF_s(M^s, x, \phi)$
Assume $(K, x) \models AG\,EF\,\phi$ holds. By CTL semantics (Definition 6.31):
$\forall r \in Runs_{ks}(K, x), \forall i \geq 0 : \exists r' \in Runs_{ks}(K, r[i]), \exists j \geq 0 : r'[j] \models \phi$

We need to show that $AGEF_s(M^s, x, \phi)$ holds, i.e.:
$\forall x' \in X_{M^s}, \forall w \in \Sigma^*_{M^s}:$
$\Big((x \xrightarrow{w} x') \implies \big(\exists v \in \Sigma^*_{M^s}, \exists x_\phi \in X_{M^s} : (x' \xrightarrow{v} x_\phi) \wedge (x_\phi \models \phi)\big)\Big)$

Consider a state $x' \in X_{M^s}$ and a word $w \in \Sigma^*_{M^s}$ such that $x \xrightarrow{w} x'$ is possible in M^s. By construction of K (Definition 6.27), this corresponds to a path from x to x' in K. Furthermore, since we assume M^s is total, we can reason about infinite runs of K. Thus:
$\exists r \in Runs_{ks}(K, x), \exists i \geq 0 : r[i] = x'$

Since $(K, x) \models AG\,EF\,\phi$ holds, from state $r[i]$ there must exist a path to a state $r'[j]$ ($j \geq 0$) where ϕ holds:
$\exists j \geq 0 : r'[j] \models \phi$

Consider $x_\phi := r'[j]$. Then the path from state $r[i] = x'$ to state $r[j] = x_\phi$ in K corresponds (by Definition 6.27) to a sequence of transitions in M^s. This sequence can be represented by a word $v \in \Sigma^*_{M^s}$ such that $x' \xrightarrow{v} x_\phi$ is possible in M^s.

From $r'[j] = x_\phi$ and $r'[j] \models \phi$ follows:
$x_\phi \models \phi$

Combining these results:
$\exists v \in \Sigma^*_{M^s}, \exists x_\phi \in X_{M^s} : (x' \xrightarrow{v} x_\phi) \wedge (x_\phi \models \phi)$

Since this holds for arbitrary $x' \in X_{M^s}$ and $w \in \Sigma^*_{M^s}$ with $x \xrightarrow{w} x'$, we conclude:
$\forall x' \in X_{M^s}, \forall w \in \Sigma^*_{M^s}$:
$\Big((x \xrightarrow{w} x') \implies \big(\exists v \in \Sigma^*_{M^s}, \exists x_\phi \in X_{M^s} : (x' \xrightarrow{v} x_\phi) \wedge (x_\phi \models \phi)\big)\Big)$

By Definition 9.7, this is exactly $AGEF_s(M^s, x, \phi)$. Thus:
$\Big((K, x) \models AG\,EF\,\phi\Big) \implies AGEF_s(M^s, x, \phi)$

Having established both directions, we conclude that the equivalence holds. □

The function EF_s defined in Definition 9.6 will be used in Theorem 9.13, where we show that if $EF_s\phi$ does not hold for a state in the specification, then there exists an uncontrollable transition to a blocking state. Furthermore, the equivalence between $AGEF_s$ and the CTL formula $AG\,EF$, established in Theorem 9.8, will be used in Theorem 9.15 to prove that $AG\,EF\,\phi$ is enforced in the corresponding Kripke structure of the controlled system.

9.3.2 Unsatisfiablility Of ϕ Causes Blocking States

In this subsection, we prove that when $EF_s\phi$ does not hold for a state in the specification, there exists an uncontrollable transition to a blocking state. This is a crucial building block in the enforcement of $AG\,EF\,\phi$ in the controlled system (Theorem 9.15). To establish this crucial building block we first have to lay its foundation in the prove of several lemmas. We will explain the usage of the consequent lemmas before stating them.

The first lemma demonstrates how $\neg EF_s$ is rewritten into a more suitable form to prove Theorem 9.13.

Lemma 9.9 (Not reachable)**.** *Consider a SEFA M, a state $x \in X$, and a predicate ϕ. Then, $\neg EF_s(M, x, \phi)$ can be rewritten to mean, for all states reachable from x, ϕ does not hold. Formally:*
$\neg EF_s(M, x, \phi) = \forall w \in \Sigma^*_M, \forall x_\phi \in X_M : \Big((x \xrightarrow{w} x_\phi) \implies (x_\phi \models \neg\phi)\Big)$

Proof.

$$\begin{aligned}
\neg EF_s(M,x,\phi) &\equiv \neg\exists w \in \Sigma_M^*, \exists x_\phi \in X_M : \big((x \xrightarrow{w} x_\phi) \land (x_\phi \models \phi)\big)\\
&\equiv \forall w \in \Sigma_M^*, \forall x_\phi \in X_M : \neg\big((x \xrightarrow{w} x_\phi) \land (x_\phi \models \phi)\big)\\
&\equiv \forall w \in \Sigma_M^*, \forall x_\phi \in X_M : \Big(\neg(x \xrightarrow{w} x_\phi) \lor \neg(x_\phi \models \phi)\Big)\\
&\equiv \forall w \in \Sigma_M^*, \forall x_\phi \in X_M : \Big((x \xrightarrow{w} x_\phi) \implies \neg(x_\phi \models \phi)\Big)\\
&\equiv \forall w \in \Sigma_M^*, \forall x_\phi \in X_M : \Big((x \xrightarrow{w} x_\phi) \implies (x_\phi \models \neg\phi)\Big)
\end{aligned}$$

□

The next lemma establishes that t when $\neg\phi$ necessarily transitions to an unmarked state of R_ϕ. This property is important in tying the unsatisfiability of ϕ in a certain state, to being a blocking state.

Lemma 9.10 (t transitions to the unmarked state when $\neg\phi$ holds)**.** *If the current state does not satisfy ϕ (i.e. $\neg\phi$ holds), then from both locations of R_ϕ, t necessarily transitions to the unmarked location q_{ng}.*

Proof. By construction of R_ϕ (Definition 8.1), the transitions are

$$(q_g,\, t,\, true,\, false,\, q_{ng}), \qquad (q_{ng},\, t,\, \phi,\, false,\, q_g), \qquad (q_{ng},\, t,\, \neg\phi,\, false,\, q_{ng}).$$

From q_g, there is only the t-transition with guard *true* to q_{ng}, so $q_g \xrightarrow{t} q_{ng}$ necessarily follows. From q_{ng}, when ϕ does not hold, only the t-transition to q_{ng} with guard $\neg\phi$ is enabled. Hence, in both locations, t transitions or self-loops necessarily to the unmarked state q_{ng} when ϕ does not hold. □

In the following lemma, we establish that when the marker predicate of R_ϕ does not hold, the marker predicate of the whole specification does not hold. And consequently, if a state is blocking with respect to R_ϕ, it is blocking for $M \| R_\phi$ as a whole.

Lemma 9.11 (Marked predicate implication)**.** *For the composed automaton $M' := M \parallel R_\phi$ the marking predicate is $p_{m,M'} = p_{m,M} \land p_{m,R_\phi}$. Consequently, $\neg p_{m,R_\phi} \implies \neg p_{m,M'}$.*

Proof. Parallel composition in this construction conjuncts marker predicates (see Definition 8.2.2). Hence, by definition, $p_{m,M'} = p_{m,M} \land p_{m,R_\phi}$. If p_{m,R_ϕ} is false, then the conjunction $p_{m,M} \land p_{m,R_\phi}$ is false, which yields $\neg p_{m,R_\phi} \Rightarrow \neg p_{m,M'}$. □

In the next lemma we establish that when R_ϕ is in its unmarked location, the path to its marked state necessarily includes a state where ϕ holds. This property is crucial in influencing synthesis in such a manner that ϕ must always be satisfiable in the controlled system.

Lemma 9.12 (ϕ required for marking of R_ϕ)**.** *From the unmarked location q_{ng} of R_ϕ, ϕ must hold in order to transition to the marked location q_g.*

Proof. By the transition set of R_ϕ (Definition 8.1), the only transition from q_{ng} to q_g is the t-transition with guard ϕ: $(q_{ng},\, t,\, \phi,\, false,\, q_g)$. □

With the preceding lemmas established, we can now tie them together to prove the main theorem of this subsection. The main theorem states that when $EF_s\phi$ does not hold for a state in the specification, there exists an uncontrollable transition to a blocking state. In Theorem 9.15 we will use this property to prove that $AG\,EF\,\phi$ is enforced in the controlled system.

Theorem 9.13 (Blocking states). *Consider a specification M, a predicate ϕ over the state of M, a requirement $R_\phi := R_{AGEF\phi}(M, \phi)$, a composed specification $M' := M \| R_\phi$, and a SEFA $M'^s := ToSEFA(M')$. Recall that $t \in \Sigma_{u,R_\phi}$ is the only and uncontrollable event of R_ϕ. Then, if $EF_s(M'^s, x, \phi)$ is violated at state x (no state reachable from x satisfies ϕ), means every state x_b reached from x via the t-transition is necessarily a blocking state, $B(M'^s, x_b)$. Formally:*

$$\forall x \in X_{M'^s} : \neg EF_s(M'^s, x, \phi) \implies \Big(\forall x_b \in X_{M'^s} : (x \xrightarrow{t} x_b) \implies B(M'^s, x_b)\Big)$$

Proof. We will focus on $\neg EF_s(M'^s, x, \phi)$ and assume it holds, as otherwise the implication is trivially satisfied.
By Lemma 9.9:
$\neg EF_s(M'^s, x, \phi) = \forall w \in \Sigma^*_{M'^s}, \forall x_\phi \in X_{M'^s} : (x \xrightarrow{w} x_\phi) \implies (x_\phi \models \neg\phi)$.

This means all reachable states from x do not satisfy ϕ. Since ϵ is a possible value of w (by $\epsilon \in \Sigma^*_{M'^s}$), this means x itself does not satisfy ϕ (as $x \xrightarrow{\epsilon} x$ by Definition 6.3). Hence:
$\neg EF_s(M'^s, x, \phi) = \forall w \in \Sigma^*_{M'^s}, \forall x_\phi \in X_{M'^s} : (x \models \neg\phi) \wedge \Big((x \xrightarrow{w} x_\phi) \implies (x_\phi \models \neg\phi)\Big)$

We add $t \in \Sigma_{u,R_\phi}$ to w: $t \cdot w$. The addition of t must be possible as it is an uncontrollable event, it is always enabled (Proposition 8.2), and it interleaves with transitions of M^s (by Definition 8.2.2). Therefore:
$\forall w \in \Sigma^*_{M'^s}, \forall x', x_\phi \in X_{M'^s} : (x \models \neg\phi) \wedge \Big((x \xrightarrow{t} x' \xrightarrow{w} x_\phi) \implies (x_\phi \models \neg\phi)\Big)$

As x does not satisfy ϕ, t transitions to an unmarked state of the R_ϕ component of M'^s (by Lemma 9.10). Thus:
$\forall w \in \Sigma^*_{M'^s}, \forall x', x_\phi \in X_{M'^s} :$
$(x \models \neg\phi) \wedge \Big((x \xrightarrow{t} x' \xrightarrow{w} x_\phi) \implies (\ (x' \models \neg p_{m,R_\phi}) \wedge (x_\phi \models \neg\phi)\ )\Big)$

By Lemma 9.12, ϕ must hold in order for the R_ϕ component of M'^s to be marked again. Therefore:
$\forall w \in \Sigma^*_{M'^s}, \forall x', x_\phi \in X_{M'^s} : (x \models \neg\phi) \wedge \Big((x \xrightarrow{t} x' \xrightarrow{w} x_\phi) \implies (x_\phi \models \neg p_{m,R_\phi})\Big)$

By Lemma 9.11, when the marker predicate of R_ϕ does not hold, the combined marker predicate p_{m,M'^s} does not hold. Consequently:
$\forall w \in \Sigma^*_{M'^s}, \forall x', x_\phi \in X_{M'^s} : (x \models \neg\phi) \wedge \Big((x \xrightarrow{t} x' \xrightarrow{w} x_\phi) \implies (x_\phi \models \neg p_{m,M'^s})\Big)$

This can be simplified to:
$\forall w \in \Sigma^*_{M'^s}, \forall x', x_\phi \in X_{M'^s} : (x \xrightarrow{t} x') \implies \Big(x' \xrightarrow{w} x_\phi \implies (x_\phi \models \neg p_{m,M'^s})\Big)$

Let us rewrite this formula as follows:
$\forall x' \in X_{M'^s} : (x \xrightarrow{t} x') \implies$
$\Big(\forall w \in \Sigma^*_{M'^s}, \forall x_\phi \in X_{M'^s} : x' \xrightarrow{w} x_\phi \implies (x_\phi \models \neg p_{m,M'^s})\Big)$

Recall the definition of blocking states (Definition 6.22):
$B(M, x) := \forall s \in \Sigma^*, \forall x' \in X : (x \xrightarrow{s} x') \implies (x' \not\models p_m)$.

We can plug the definition of blocking states into the rewritten formula as follows:
$\forall x' \in X_{M'^s} : (x \xrightarrow{t} x') \implies B(M'^s, x')$

Thus we can conclude:
$\forall x \in X_{M'^s} : \neg EF_s(M'^s, x, \phi) \implies \Big(\forall x_b \in X_{M'^s} : (x \xrightarrow{t} x_b) \implies B(M'^s, x_b)\Big)$ □

We have established that a violation of $EF_s\phi$ for a state in the specification leads to an uncontrollable transition to blocking states. Synthesis must prevent this from happening in all reachable states by restricting the specification. This thus translates the $EF_s\phi$ property to an $AGEF_s\phi$ property for all initial states of the controlled system. In the following subsection, we will elaborate further on this enforcement.

9.3.3 $AG\ EF\,\phi$ is Enforced in the Controlled System

Finally, in the subsequent theorem we prove M'^s enforces $AG\ EF\,\phi$ in the controlled system. But first, we prove that a controlled system derived from a specification with R_ϕ is total, as required for CTL semantics. In the proof of Theorem 9.15, we will combine the bridge between CTL and SEFA semantics we have established in Section 9.3.1, together with Theorem 9.13 to demonstrate that synthesis will restrict the specification such that $AG\ EF\,\phi$ holds for all initial states of the controlled system.

In the next lemma we prove that the controlled system including R_ϕ is total (Definition 6.8). Recall that totality ensures all runs can be extended indefinitely, which is essential for CTL semantics.

Lemma 9.14 (Controlled system is total)**.** *Consider a specification M, a predicate ϕ over the state of M, a requirement $R_\phi := R_{AGEF\phi}(M, \phi)$, a composed specification $M' := M \| R_\phi$, and a controlled system $C' := M' \|\, Synth(M', p_{f,M'})$. Then, C' is total (Definition 6.8), meaning there always exists an enabled outgoing transition for every state.*

Proof. In Proposition 8.2 we have established that transition $t \in \Sigma_{R_\phi}$ is always enabled. t in fact is uncontrollable, i.e., $t \in \Sigma_{u,M'}$. By the definition of controllability (Definition 6.20), uncontrollable transitions cannot be restricted in the controlled system, when they are unrestricted in the specification. Hence the controlled system C' remains total. □

Theorem 9.15 ($AG\ EF\,\phi$ is enforced)**.** *Consider a specification EFA M, a predicate ϕ over the state of M, a requirement automaton $R_\phi := R_{AGEF\phi}(\phi, M)$, a combined specification $M' := M \parallel R_\phi$, the forbidden state predicate p_f for M', and a controlled system $C' := M' \|\, Synth(ToSEFA(M'), p_f)$. Then, $\forall x_0 \in X^0_{C'} : (KS(C', L(C')), x_0) \models AG\ EF\,\phi$ holds.*

Proof. By Definition 9.7:
$\Big(\forall x_0 \in X^0_{C'} : (KS(C', L(C')), x_0) \models AG\ EF\,\phi\Big) = \Big(\forall x_0 \in X^0_{C'} : AGEF_s(C', x_0, \phi)\Big)$
Furthermore, in Lemma 9.14, we have established that C' is total, meaning all its runs can be extended indefinitely, which is required to preserve CTL semantics.

We prove $AGEF_s(C', x_0, \phi)$ by contradiction. Assume there exists an initial state $x_0 \in X^0_{C'}$ such that $AGEF_s(C', x_0, \phi)$ does not hold. Then we derive:
$\exists x_0 \in X^0_{C'} : \exists x' \in X_{C'}, \exists w \in \Sigma^*_{C'} :$
$\Big((x_0 \xrightarrow{w} x') \wedge \Big(\forall v \in \Sigma^*_{C'}, \forall x_\phi \in X_{C'} : (\ (x' \xrightarrow{v} x_\phi) \implies (x_\phi \models \neg\phi)\ )\Big)\ \Big)$

By lemma 9.9:
$\forall v \in \Sigma^*_{C'}, \forall x_\phi \in X_{C'} : (\ (x' \xrightarrow{v} x_\phi) \implies (x_\phi \models \neg\phi)\ ) \equiv \neg EF_s(C', x', \phi)$

Hence, $\neg EF_s(C', x', \phi)$ can be inserted to simplify the formula to:
$\exists x_0 \in X^0_{C'} : \exists x' \in X_{C'}, \exists w \in \Sigma^*_{C'} : \Big((x_0 \xrightarrow{w} x') \wedge \neg EF_s(C', x', \phi)\Big)$

By Theorem 9.13, when $\neg EF_s(C', x', \phi)$ holds, $t \in \Sigma_{u,R_\phi}$ transitions to a blocking state. In Proposition 8.2 we have established that such a transition is always possible. Combining both aspects we derive:
$\exists x'' \in X_{C'} : (x' \xrightarrow{t} x'') \wedge B(C', x'')$

By the definition of controllability (Definition 6.20), uncontrollable transitions cannot be restricted when they are unrestricted in the specification. Hence, the transition $(x' \xrightarrow{t} x'')$ must also exist in C'. Therefore:
$\exists x_0 \in X^0_{C'} : \exists x'' \in X_{C'}, \exists w \in \Sigma^*_{C'} : \Big((x_0 \xrightarrow{wt} x'') \wedge B(C', x'')\Big)$

As result of the truth assumption we derived the existence of a path to a blocking state. However, this is a direct contradiction of Lemma 6.23, which states:
$\forall x \in States(runs(C, L(C))) : \neg B(C, x)$

Consequently, we conclude the theorem holds:
$\forall x_0 \in X^0_{C'} : (KS(C', L(C')), x_0) \models AG\ EF\ \phi$ □

To summarize the non-blockingness section of the $R_{AGEF\phi}$ correctness proof, we have established that the CTL formula $AG\ EF\ \phi$ can be enforced in reachable states of the controlled system when requirement automata R_ϕ, generated by $R_{AGEF\phi}$, are composed with the specification. In Section 9.3.1, we have linked the semantics of CTL operands to corresponding SEFA functions. EF_s is such a SEFA function which we used in Section 9.3.2 to prove that any violation of $EF_s\phi$ in a specification leads to an uncontrollable transition to a blocking state. Synthesis must prevent this by restricting the specification. Our approach thus utilizes the controllability (Definition 6.20) and non-blockingness (Definition 6.21) properties of supervisory control theory to influence synthesis such that $AG\ EF\ \phi$ holds for all initial states of the controlled system.

9.4 Maximally Permissive

The final property of synthesis that must be respected is maximal permissiveness. We prove that the addition of R_ϕ preserves this property, as it only influences synthesis to enforce $AG\ EF\ \phi$ (Definition 6.31), which is precisely the goal. R_ϕ does not directly impact safety and controllability, as proven in Theorem 9.3 and Theorem 9.4 respectively. In Theorem 9.15, we have established that R_ϕ influences non-blockingness to ensure $AG\ EF\ \phi$ holds in the controlled system. Synthesis will ensure that the controlled system is maximally permissive, but since non-blockingness is influenced by R_ϕ, we must ensure that this influence precisely achieves our goal. In Theorem 9.13, we have proven that when EF_s with respect to ϕ does not hold t transitions to a blocking state. In this section we will prove the reverse implication, with a slight modification. We will prove that when t transitions to a blocking state, then either EF_s with respect to ϕ does not hold, or all reachable states satisfying ϕ are blocking states with respect to the original specification EFA, that is, without R_ϕ. Finally, as this is exactly what we aim to achieve, we conclude maximal permissiveness is preserved which concludes the correctness proof.

Theorem 9.16 (Non-blockingness preservation)**.** *Consider an EFA M, a predicate ϕ over the state of M, a requirement automaton $R_\phi := R_{AGEF\phi}(\phi, M)$, a SEFA $M^s :=$ ToSEFA(M), and a combined SEFA $M'^s :=$ ToSEFA$(M \parallel R_\phi)$. Then, for all states*

x in the combined SEFA M'^s, if t transitions to a blocking state, then either EF_s with respect to ϕ does not hold, or all reachable states from x satisfying ϕ are blocking states with respect to the original SEFA M^s. Formally:
$\forall x \in X_{M'^s} : \Big(\forall x_b \in X_{M'^s} : (x \xrightarrow{t} x_b) \implies B(M'^s, x_b)\Big) \boxed{\implies}$
$\Big(\neg EF_s(M'^s, x, \phi) \vee \big(\ \forall x' \in X_{M'^s}, w \in \Sigma^*_{M'^s} : x \xrightarrow{w} x' \implies (x' \models \phi \implies B(M^s, x'))\ \big)\Big)$

Proof. The proof is by contradiction. Assume that there exists a state x such that the left hand side of the boxed implication holds but that the conclusion (right hand side) does not follow. Formally:
$\exists x \in X_{M'^s} : \Big(\forall x_b \in X_{M'^s} : (x \xrightarrow{t} x_b) \implies B(M'^s, x_b)\Big) \wedge$
$\Big(EF_s(M'^s, x, \phi) \wedge \big(\ \exists x' \in X_{M'^s}, w \in \Sigma^*_{M'^s} : x \xrightarrow{w} x' \wedge (x' \models \phi \wedge \neg B(M^s, x'))\ \big)\Big)$

By the definition of parallel composition (Definition 6.10) and the construction of M'^s, the marker predicate p_{m,M'^s} of M'^s is $p_{m,M} \wedge p_{m,R_\phi}$. By the definition of blocking states (Definition 6.22), $B(M'^s, x_b)$ means there does not exist a path from x_b to a marked state of M'^s. Since the marker predicate of M'^s is a conjunction, this means there does not exist a path from x_b to a state where both $p_{m,M}$ and p_{m,R_ϕ} hold. Thus, we can rewrite the left hand side of the boxed implication as follows:
$\exists x \in X_{M'^s} : \Big(\forall x_b \in X_{M'^s} : x \xrightarrow{t} x_b \implies$
$\big(\forall x'_b \in X_{M'^s}, w_b \in \Sigma^*_{M'^s} : x_b \xrightarrow{w_b} x'_b \implies (x'_b \not\models p_{m,M} \wedge p_{m,R_\phi})\big)\Big) \boxed{\wedge}$
$\Big(EF_s(M'^s, x, \phi) \wedge \big(\ \exists x' \in X_{M'^s}, w \in \Sigma^*_{M'^s} : x \xrightarrow{w} x' \wedge (x' \models \phi \wedge \neg B(M^s, x'))\ \big)\Big)$

On the right hand side of the boxed conjunction, we have that there exists a reachable state x' from x such that x' satisfies ϕ and is not blocking with respect to M^s. By the definition of blocking states (Definition 6.22), $\neg B(M^s, x')$ means there exists a path from x' to a marked state of M^s, i.e., $p_{m,M}$ holds. Thus:
$\exists x \in X_{M'^s} : \Big(\forall x_b \in X_{M'^s} : x \xrightarrow{t} x_b \implies$
$\big(\forall x'_b \in X_{M'^s}, w_b \in \Sigma^*_{M'^s} : x_b \xrightarrow{w_b} x'_b \implies (x'_b \not\models p_{m,M} \wedge p_{m,R_\phi})\big)\Big) \boxed{\wedge}$
$\Big(EF_s(M'^s, x, \phi) \wedge \big(\ \exists x' \in X_{M'^s}, w \in \Sigma^*_{M'^s} : x \xrightarrow{w} x' \wedge (x' \models \phi \wedge$
$\big(\exists x'' \in X_{M'^s}, w'' \in \Sigma^*_{M'^s} : x' \xrightarrow{w''} x'' \wedge x'' \models p_{m,M})\big)\ \big)\Big)$

As ϕ holds in x', either t or $t \cdot t$ transitions to a state where p_{m,R_ϕ} holds (depending on the R_ϕ component of x') by the construction of R_ϕ (Definition 8.1) and Lemma 9.12. As t interleaves with all other transitions, by construction, it does not impact the reachability of states where $p_{m,M}$ holds. Thus, there exists a path consisting of one or two t-transitions after which p_{m,R_ϕ} holds from where there is a path to a state where $p_{m,M}$ holds. Therefore, there exists a path from x to a state where both $p_{m,M}$ and p_{m,R_ϕ} hold, contradicting the left hand side of the boxed conjunction. Hence, we conclude our contradictory assumption must be false and thus the theorem holds. □

As this theorem holds for **any** EFA composed with R_ϕ and all cases from R_ϕ (all t-transitions and all states of R_ϕ) have been considered, we can conclude that R_ϕ only influences synthesis as established in Theorem 9.13.

9.4.1 *AG EF ϕ* Is Enforced Maximally Permissively

The correctness of $R_{AGEF\phi}$ relies on the EFAs R_ϕ it produces based on a given predicate ϕ and specification EFA M. We consider $R_{AGEF\phi}$ "correct" when the resulting controlled system of the composition $M \parallel R_\phi$ enforces $AG\ EF\ \phi$ maximally permissively (in the corresponding Kripke structure of the controlled system, Definition 6.27). Maximal permissiveness (Definition 6.24) hinges on three properties: safety, controllability, and non-blockingness. In Theorem 9.3 we have established that safety is preserved when adding R_ϕ, meaning no additional unsafe states or transitions are directly introduced. In Theorem 9.4 we have established that R_ϕ preserves controllability with respect to M, as only an uncontrollable event t is introduced with solely interleaving transitions. In Section 9.3 we have:

1. Bridged the SEFA and CTL semantics (Theorem 9.8).
2. Established that $\neg EF_s\ \phi$ leads to an uncontrollable transition to a blocking state (Theorem 9.13).
3. Proven in Theorem 9.15 that due to the non-blockingness property of synthesis and Theorem 9.13, $EF_s\ \phi$ is enforced in all reachable states of the controlled system, i.e, $AGEF_s\ \phi$ holds in the controlled system.

Finally, in Theorem 9.16, we have established that R_ϕ only influences marking such that if any t transition leads to a blocking state, then either EF_s with respect to ϕ does not hold, or all reachable states satisfying ϕ are blocking states with respect to the original specification EFA. This aspect combined with Theorem 9.15, proves that R_ϕ only restricts non-blockingness as strictly necessary to enforce $AG\ EF\ \phi$. As this is exactly what we aim to achieve, and synthesis is maximally permissive, we conclude $R_{AGEF\phi}$ is correct.

10 Experiments

In this section, we compare the synthesis performance of two approaches for enforcing liveness requirements: the reachability requirement annotation (Section 8.1), referred to as the algorithmic approach, and the reachability requirement automaton (Section 8.2), referred to as the model-based approach.

The experimental methodology is outlined in Section 10.1, which begins with a description of the 15 benchmark models used in the experiments (Section 10.1.1). This is followed by the experimental setup (Section 10.1.2) and a justification of correctness (Section 10.1.3). After presenting the methodology, we report the experimental results in Section 10.2 and discuss them in Section 11.

10.1 Methodology

In this subsection, we outline the methodology used for the experimental setup to compare both approaches with each other and with a baseline. For this comparison, we use 15 benchmark models that we describe in the following subsection.

10.1.1 Benchmark Models

The 15 benchmark models used in the experiments either directly model real cyber-physical systems or are inspired by them. 13 of the 15 models are part of the benchmark suite shipped with the Eclipse ESCET toolkit, while the remaining two models (the generic lock case studies) are provided by RWS. Table 2 provides an overview of all benchmark models and their sources, including a short name per model that we use in subsequent tables. The case study models (benchmarks 6 and 7) do not have references, as they are generated using a proprietary tool from RWS. Table 2 is an adapted version of the one presented in [9].

Nr	**Short name**	**Full name**	**Refs**
1	adas	Advanced driver assistance system	[47]
2	agv	Automated guided vehicles	[48]
3	bcs-dynamic	Body comfort system (dynamic)	[49, 50]
4	bcs-static	Body comfort system (static)	[49, 50]
5	bridge	Bridge	[51]
6	case-study-org	Generic lock case study (original)	-
7	case-study-inv	Generic lock case study (with invariant)	-
8	cluster-tool	Cluster tool	[52]
9	festo	FESTO production line	[53]
10	mri-pss-event	MRI scanner PSS (event-based)	[54, 55]
11	mri-pss-state	MRI scanner PSS (state-based)	[55]
12	prod-cell	Production cell	[56]
13	theme-park	Theme park vehicles	[57]
14	wafer-scanner-n1	Wafer scanner ($n = 1$)	[58]
15	waterway-lock	Waterway lock	[59]

Table 2: Overview of the benchmark models. This table is an adapted version of the one presented in [9].

Below, we provide a brief description of each benchmark model. These descriptions, except the case study model descriptions (benchmarks 6 and 7), are adapted from [9].

1) The *advanced driver assistance system* (ADAS) is a safe controller for an adaptive cruise control system of the Toyota Prius Executive [47].

2) The *automated guided vehicles* (AGVs) model five vehicles serving a manufacturing work cell. The vehicles travel on fixed circular routes through shared zones, where collisions must be avoided [48].

3+4) The *body comfort system* (BCS) is a product line originally from the automotive industry. There is a *dynamic* version (benchmark 3) that allows reconfiguration during execution, and a *static* version (benchmark 4) that does not [49, 50].

5) The Algera *bridge* is a movable bridge over the Hollandse IJssel river in the Netherlands. The synthesized controller ensures safe operation, even with faulty sensors or broken actuator connections [51].

6+7) The *generic lock case study* models a generic waterway lock used to raise and lower boats between waters of different levels. This model is the case study provided by RWS that motivated this research. Benchmark 6 is the original version, while benchmark 7 includes the requirement invariant described in Section 5. The liveness requirements added in these models ensure that a waterway lock gate is openable and closable, as required by RWS, see Section 5.

8) The *cluster tool* is a wafer processing system with nine processing chambers, four transportation robots, and three one-slot buffers. The synthesized controller guarantees continuous wafer processing without blocking [52].

9) The *FESTO production line* consists of six connected stations that distribute, handle, test, buffer, drill, and store products. The supervisor ensures safe, correct, and efficient operation [53].

10+11) The *patient support system* (PSS) positions a patient in an MRI scanner using a movable table. The synthesized supervisor ensures safe operation to prevent damage to the scanner and patient [54, 55]. Benchmark 10 is event-based, while benchmark 11 uses state-based invariants.

12) The *production cell* consists of a stock, feed belt, elevating rotary table, robot, press, deposit belt, and crane. These components execute asynchronously but synchronize for cooperation and safety [56].

13) The Multimover is a flexible *theme park vehicle* that follows an electrical wire integrated in the floor. Floor codes provide information about the track and control the vehicle's route, speed, and music. The synthesized supervisor ensures safe operation and prevents collisions [57].

14) The *wafer scanner* is a lithography machine used in integrated circuit production. Wafers undergo pre-exposure steps and are measured and exposed on one of two chucks. Dummy wafers are inserted to maintain the water film required for immersion lithography. The model for $n = 1$ includes one production wafer and two dummy wafers [58].

15) Lock III is a *waterway lock* in Tilburg, the Netherlands. Gates seal off the chamber, paddles empty it, and culverts fill it. The supervisor ensures safe and correct operation [59].

10.1.2 Experimental Setup

For all 15 models, we add the same liveness requirements (in the form of $AG\ EF\ \phi$) using both approaches and measure synthesis performance in a platform-independent way using Binary Decision Diagram (BDD) operations and maximum node count as proxies for speed and memory usage, respectively [44]. Both approaches are additionally compared, as a baseline, to the original 15 models without extra liveness requirements. Apart from performance, we additionally measure the resulting state-space sizes of the controlled systems. This allows us to observe whether the liveness requirements affect the state-space size, and additionally, the state-space size helps us to verify our hypothesis described in the subsequent subsection. All experiments are executed on a machine with a 2.80 GHz Intel Core i7-1165G7 processor and 32 GB RAM, using the default settings of CIF's symbolic synthesis tool from Eclipse ESCET v10.0. For reproducibility purposes, all models and experiment scripts are available at [60].

10.1.3 Correctness Validation

In Section 9, we proved the correctness of the model-based approach, hence we use it to validate the correct implementation of the algorithmic approach. This validation is performed by comparing the controlled systems resulting from both approaches on ten smaller models. The comparison starts by projecting away the t-transitions of the model-based approach, as these transitions are not present in the algorithmic approach. Afterwards, the *cif2mcrl2* function [61] is used to translate both controlled systems to the mCRL2 specification language [18]. This translation adds a marked action enabled as self-loop in all marked states of the controlled systems, enabling the verification of marked languages. Subsequently, mCRL2 is used to test for strong trace equivalence between the two transformed controlled systems. As the RW-framework revolves around marked languages and due to the addition of marked actions, strong trace equivalence is a suitable equivalence notion for this validation. All tests confirmed that both controlled systems are strongly trace equivalent. These tests are available at [60]. This validation process does not guarantee the correctness of the algorithmic approach, but it does provide confidence in its correct implementation.

Since the model-based approach adds two locations with solely interleaving t-transitions per liveness requirement, every added liveness requirement should double the state-space of the controlled system, compared to the algorithmic approach. The reason for this can be observed in the parallel composition of the model-based approach (Section 8.2.2) with a specification. In this composition, it can be seen that the addition of two locations multiplies the state-space size similarly, and since no synchronizing transitions are added, no states can be merged either. Furthermore, as the model-based approach only includes uncontrollable transitions, none of its states can be removed during synthesis. Therefore, we hypothesize that the controlled system state-space size is multiplied by two per added liveness requirement relative to the algorithmic approach. We will verify this hypothesis in the discussion section (Section 11).

10.2 Results

Table 10.2 displays the experimental results. Both the number of BDD operations and the maximum number of nodes are presented in normalized form, where the lowest value for each model is set to 1.0, and other values are expressed as ratios to this baseline. #Liveness indicates the number of liveness requirements added to each model. Original refers to the baseline models without added liveness requirements, Algorithmic refers to the models with liveness requirements added using the algorithmic approach, Model-based refers to the models with liveness requirements added using the model-based approach. The case study models (benchmarks 6 and 7) are highlighted in bold and surrounded by horizontal lines, while the bottom row provides average values across all models for each metric. Dashes ("-") indicate that synthesis resulted in an empty supervisor, thus yielding a state-space size of 0. The results are further discussed in Section 11.

			Number of BDD operations			Maximum number of nodes			Controlled system state-space		
Nr	**Model name**	**#Liveness**	**Original**	**Algorithmic**	**Model-based**	**Original**	**Algorithmic**	**Model-based**	**Original**	**Algorithmic**	**Model-based**
1	adas	4	1.000	1.905	1.650	1.000	1.000	1.045	2.72×10^{10}	2.72×10^{10}	4.35×10^{11}
2	agv	5	1.000	1.834	1.299	1.584	1.000	1.417	2.88×10^{04}	4.73×10^{03}	1.51×10^{05}
3	bcs-dynamic	5	1.000	1.151	1.051	1.000	1.022	1.280	4.23×10^{20}	3.85×10^{20}	1.23×10^{22}
4	bcs-static	5	1.210	1.504	1.000	1.449	1.067	1.000	3.96×10^{17}	5.38×10^{16}	1.72×10^{18}
5	bridge	1	1.000	1.965	1.972	1.000	1.037	1.040	5.92×10^{33}	-	-
6	**case-study-org**	**2**	**1.000**	**1.953**	**2.640**	**1.000**	**1.342**	**1.130**	$\mathbf{1.81 \times 10^{57}}$	$\mathbf{1.47 \times 10^{57}}$	$\mathbf{5.89 \times 10^{57}}$
7	**case-study-inv**	**2**	**1.280**	**1.562**	**1.000**	**1.488**	**1.008**	**1.000**	$\mathbf{1.21 \times 10^{57}}$	-	-
8	cluster-tool	4	1.000	1.353	1.019	1.000	1.000	1.006	1.87×10^{07}	1.87×10^{07}	3.00×10^{08}
9	festo	18	1.000	1.414	1.498	1.000	1.000	1.104	1.06×10^{28}	1.06×10^{28}	2.77×10^{33}
10	mri-pss-event	2	1.000	1.296	2.932	1.003	1.000	1.266	2.36×10^{11}	1.26×10^{11}	5.06×10^{11}
11	mri-pss-state	4	1.000	1.891	2.004	1.000	1.074	1.038	1.71×10^{05}	1.43×10^{05}	2.28×10^{06}
12	prod-cell	4	1.000	1.321	1.757	1.000	1.000	1.711	5.24×10^{08}	5.24×10^{08}	8.38×10^{09}
13	theme-park	9	1.000	2.100	2.438	1.000	1.000	1.182	6.02×10^{07}	6.02×10^{07}	3.08×10^{10}
14	wafer-scanner-n1	27	1.000	2.918	169.855	1.589	1.000	2.405	1.14×10^{07}	1.04×10^{06}	1.40×10^{14}
15	waterway-lock	10	1.000	1.389	1.410	1.000	1.000	1.002	5.96×10^{32}	5.96×10^{32}	6.10×10^{35}
	Average	**7.620**	**1.033**	**1.704**	**12.902**	**1.141**	**1.037**	**1.242**	$\mathbf{2.02 \times 10^{56}}$	$\mathbf{1.13 \times 10^{56}}$	$\mathbf{4.53 \times 10^{56}}$

Table 3: Results for the number of BDD operations, maximum number of nodes, and state-space for the original models, model-based approach, and algorithmic approach.

11 Discussion

In this section, we discuss the experimental results presented in Table 10.2. We compare both approaches in terms of synthesis performance and controlled system state-space size. Furthermore, we verify our hypothesis regarding the relationship between the state-space sizes of both approaches. For the algorithmic approach, we additionally compare the resulting state-spaces with the baseline to detect whether the additional liveness requirements have affected the controlled system. This cannot be (directly) done for the model-based approach, as it adds states to the specification by construction.

From Table 10.2 we observe that the algorithmic approach requires, on average, 1.704 times more BDD operations and 1.037 times more maximum number of nodes than the best performing approach per model. The model-based approach, on the other hand, requires 12.902 times more BDD operations and 1.242 times more maximum number of nodes than the best performing approach per model. This indicates that the algorithmic approach is on average 7.6 times faster in terms of BDD operations and 1.2 times more efficient in terms of maximum number of nodes than the model-based approach.

However, the algorithmic approach does not always outperform the model-based approach. In terms of number of BDD operations, the model-based approach outperforms the algorithmic approach in the benchmarks 1, 2, 3, 4, 7, and 8. In terms of maximum number of nodes, the model-based approach outperforms the algorithmic approach in benchmarks 4, 6, 7, and 11. The exact reasons for these outliers require further investigation, but we hypothesize that variable ordering effects in BDDs play a significant role.

We additionally analyzed the relationship between the number of added liveness requirements and the performance overhead. For this analysis, we calculate the average increase in number of BDD operations and maximum number of nodes per liveness requirement for both approaches. From these calculations, we omit models that result in an empty supervisor for either approach (benchmarks 5 and 7). The algorithmic approach shows an average increase of 13.5% in BDD operations and a decrease of 1.3% in maximum number of nodes per added liveness requirement. The decrease in maximum number of nodes for the algorithmic approach can be explained by the fact that it does not add states to the specification, but rather removes states by enforcing liveness directly on the existing states. The model-based approach shows an average increase of 68.5% in BDD operations and an increase of 1.3% in maximum number of nodes per added liveness requirement. This further confirms that on average the algorithmic approach has substantially lower performance overhead when adding liveness requirements.

Several outliers in the data deserve discussion. First, the generic lock with invariant model (benchmark 7) results in an empty supervisor for both approaches. This was expected, as RWS identified states in the original model that could never satisfy the added liveness requirements. An empty supervisor directly indicates to RWS that the invariant is too strict and needs to be relaxed. Second, it is remarkable to observe that even the state-space of the generic lock original model (benchmark 6) decreases (see the state-space in the column “Algorithmic”) when adding liveness requirements using the algorithmic approach. This implies that the challenge posed by RWS is already present in the original model, and that adding the liveness requirements helps to eliminate states from which the waterway lock gate cannot be opened or closed. This observation further supports the value of adding additional liveness requirements to the RWS case study model and to models in general. No effects on the controlled system state-space were observed for adas, cluster-tool, festo, prod-cell, theme-park, and waterway-lock (benchmarks 1, 8, 9, 12, 13, and 15), meaning the original controlled system already satisfied the added liveness requirements. Another interesting observation is that the

Nr	**Model**	**Algorithmic**	**Model-based**	**Exponent**
1	adas	2.72×10^{10}	4.35×10^{11}	4
2	agv	4.73×10^{03}	1.51×10^{05}	5
3	bcs-dynamic	3.85×10^{20}	1.23×10^{22}	5
4	bcs-static	5.38×10^{16}	1.72×10^{18}	5
5	bridge	0	0	–
6	case-study-org	1.47×10^{57}	5.89×10^{57}	2
7	case-study-inv	0	0	–
8	cluster-tool	1.87×10^{07}	3.00×10^{08}	4
9	festo	1.06×10^{28}	2.77×10^{33}	18
10	mri-pss-event	1.26×10^{11}	5.06×10^{11}	2
11	mri-pss-state	1.43×10^{05}	2.28×10^{06}	4
12	prod-cell	5.24×10^{08}	8.38×10^{09}	4
13	theme-park	6.02×10^{07}	3.08×10^{10}	9
14	wafer-scanner-n1	1.04×10^{06}	1.40×10^{14}	27
15	waterway-lock	5.96×10^{32}	6.10×10^{35}	10

Table 4: Verification that the model-based approach state-space equals the algorithmic approach state-space multiplied by 2 to the power of the number of liveness requirements.

model-based approach results in the fewest BDD operations for the body comfort system static model (benchmark 4). This can likely be explained by the influence of variable ordering on BDD performance. Heuristic algorithms are used to determine a good variable ordering for BDDs, and it is possible that the addition of the requirement automaton in this case led to a more optimal variable ordering due to the heuristic nature of these algorithms. For more information on BDD variable ordering in CIF, we refer to [9]. Obviously, the model-based approach could change the variable ordering of all models compared to the original models. This cannot be the case for the algorithmic approach, as it does not influence the heuristic variable ordering. We did not account for effects on variable ordering in our analysis and leave this for future work. The wafer scanner model (benchmark 14) represents another notable outlier, particularly for the model-based approach. In this case, the model-based approach requires 169.855 times more BDD operations and 2.405 times more maximum number of nodes than the original model. This substantial overhead is likely because this model is already the most demanding in terms of both BDD metrics. Adding additional liveness requirements, especially for the model-based approach, which increases the state-space size, further increases this demand significantly.

In Section 10.1.3, we posed the hypothesis that the state-space size of the model-based approach equals the state-space size of the algorithmic approach multiplied by two to the power of the number of added liveness requirements. Table 4 confirms this relationship for all models, showing that the model-based approach state-space equals the algorithmic approach state-space multiplied by 2^{Exponent}, where Exponent is equal to the number of added liveness properties. This observation further supports the correctness of both approaches.

Overall, we have observed some interesting effects on the controlled system state-space size when adding liveness requirements using the algorithmic approach. Especially for the RWS case study models (benchmarks 6 and 7), adding liveness requirements affects the state-space size, even in the original model (benchmark 6). However, the most significant finding from the experiments is the superior synthesis performance of the algorithmic approach compared to the model-based approach (on average). For this reason, the algorithmic approach has been adopted in version 10.0 of CIF.

12 Conclusion

This work started with the challenge encountered by RWS, during experimentation with state requirement invariants, in which they identified states in the controlled system of a waterway lock from where it was no longer possible to open or close a lock gate. The standard liveness capabilities of CIF in the form of marking are not adequate to capture the desired reachability of multiple states in the controlled system. Based on this shortcoming, we formulated the following research question: *How can CIF be extended to support liveness requirements that require multiple states to be reachable in the controlled system, while preserving safety, controllability, non-blockingness, and maximal permissiveness as they are currently defined?*

To answer this research question, we first formalized the desired liveness requirements of RWS to the CTL formula $AG\ EF\ \phi$, where ϕ is a state formula that captures the desired states to be reached. This way RWS would be able to specify $AG\ EF$ `open` and $AG\ EF$ `closed`, where `open` and `closed` represent states in which a lock gate is open and closed respectively, to ensure that from every reachable state in the controlled system, it is always possible to eventually open or close a lock gate. Next, we performed a literature study on existing approaches, which are compatible with CIF, to express liveness requirements in supervisory control theory. Based on this literature study we identified two promising sources with a similar approach: capture the liveness requirement $AG\ EF\ \phi$ as additional reachability search in the synthesis algorithm. Additionally, we investigated a purely model-based approach using existing constructs of CIF to express the desired liveness requirements. We believe this approach to be novel, as we did not identify any source which proposed a similar solution. Due to the novelty of this approach, we proved its correctness with respect to safety, controllability, non-blockingness, and maximal permissiveness. Using either of the two approaches, RWS' challenge is solved as it can be guaranteed that from every reachable state a lock gate can always be opened or closed, or the synthesis results in an empty supervisor. The latter is the case for the state requirement experiments RWS case study model. The additional liveness capabilities reduce required validation time as it is no longer necessary to check whether all desired states are reachable in the controlled system. More fundamentally, it strengthens the correct-by-construction notion of synthesis by providing more precise conditions under which a supervisor is considered correct.

To decide which of the two approaches is most suitable to be integrated in CIF, we performed experiments on a set of 15 benchmark models which either directly model real cyber-physical systems or are inspired by them. The experiments clearly show the superiority of the algorithmic approach in terms of synthesis speed and maximal memory usage. Therefore, this approach has been adopted in CIF v10.0 to support liveness requirements of the form $AG\ EF\ \phi$. There is a minor limitation to this approach regarding the CTL semantics of $AG\ EF\ \phi$; it does not require the controlled system to be total whereas this is required by CTL semantics. It is theoretically possible that a reachable state with no outgoing transitions could be safe and satisfy all liveness requirements, and therefore be included in the controlled system. Nevertheless, we deem this limitation not to be problematic in practice as this property also currently holds for marking in CIF.

To conclude, we have successfully extended CIF with liveness capabilities that allow RWS to express their liveness requirements and solved their challenge regarding unreachable desired states in the controlled system of the waterway lock experiment.

12.1 Future Work

In this work we have extended CIF with liveness capabilities that allow RWS to express their liveness requirements. However, there are still several directions for future work to further improve liveness support in CIF. First, the extended liveness capabilities only support a very limited subset of CTL, namely $AG\ EF\ \phi$, where ϕ is a predicate capturing desired states. Future work could investigate how to extend CIF with more expressive liveness requirements, for example by supporting different nested CTL formulas. Such formulas could for example express reactivity requirements, e.g., $AG(\text{request} \rightarrow AF\ \text{grant})$, which states that whenever a request occurs, a grant should eventually follow. But as we have described in the literature study, supporting more expressive CTL formulas comes with significant challenges regarding the uniqueness of the synthesized supervisor and thus yielding an adaptation to the definition of maximal permissiveness.

Notably, the original RWS case study model without the invariant already contained states in the controlled system from which it was not possible to open or close a certain waterway lock gate. Future work could investigate which states were removed by the added liveness requirements and why, as well as whether those states represent desirable behavior. If such states are indeed desirable, future work could investigate how to change the liveness requirements to preserve those states while still guaranteeing that the liveness requirement holds for all other reachable states in the controlled system.

Another direction would be to investigate the adoption of LTL to express liveness requirements. Such liveness requirements are stricter than CTL in the sense that LTL requires infinitely often visiting the desired states, whereas CTL only requires the desired states to be reachable. Current LTL approaches in supervisory control theory drastically impact the possible behavior in the controlled system which impacts maximal permissiveness.

Future work could also investigate fairness assumptions on uncontrollable events to express which uncontrollable transitions should be taken into account when verifying liveness requirements.

Regarding model-based approaches, while there is likely much more possible using such approaches to express liveness/temporal requirements in CIF, we recommend to focus on algorithmic approaches as we hypothesize, based on the experiments, that most model-based approaches will likely not be as efficient as a tailored algorithmic one. Nevertheless, model-based approaches could still be an alternative while there is no algorithmic approach available for a specific type of (liveness) requirement.

Furthermore, it would be interesting to investigate what the reachability requirement automaton did to the variable ordering and whether this explains why the model-based approach sometimes outperformed the algorithmic approach. Additionally, another interesting direction for future work would be to investigate whether fixing the variables of the reachability requirement automata in order and/or place would improve the performance of the model-based approach.

Lastly, we proved the correctness of the model-based approach using classical RW-concepts in the context of CIF. As CIF fully aligns with original definitions of the RW-framework we believe that the correctness of the model-based approach naturally extends to any synthesis tool aligning with the RW-framework. However, future work could formalize the correctness of the model-based approach solely in the RW-framework to prove this natural extension.

References

[1] Bram van der Sanden, Michel Reniers, Marc Geilen, Twan Basten, Johan Jacobs, Jeroen Voeten, and Ramon Schiffelers. Modular model-based supervisory controller design for wafer logistics in lithography machines. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 416–425. IEEE, 2015.

[2] Rolf JM Theunissen, Mihaly Petreczky, Ramon RH Schiffelers, Dirk A van Beek, and Jacobus E Rooda. Application of supervisory control synthesis to a patient support table of a magnetic resonance imaging scanner. *IEEE Transactions on Automation Science and Engineering*, 11(1):20–32, 2013.

[3] Wan Fokkink, Martijn Goorden, Joanna Van de Mortel-Fronczak, Ferdie Reijnen, and Jacobus Rooda. Supervisor synthesis: Bridging theory and practice. *Computer*, 55(10):48–54, 2022.

[4] FFH Reijnen, MA Reniers, JM Van de Mortel-Fronczak, and JE Rooda. Structured synthesis of fault-tolerant supervisory controllers. *IFAC-PapersOnLine*, 51(24):894–901, 2018.

[5] Mark Utting and Bruno Legeard. *Practical model-based testing: a tools approach.* Elsevier, 2010.

[6] Peter J Ramadge and W Murray Wonham. Supervisory control of a class of discrete event processes. *SIAM journal on control and optimization*, 25(1):206–230, 1987.

[7] Eclipse Foundation, Inc. Eclipse ESCET. `https://www.eclipse.dev/escet`, 2025. Eclipse, Eclipse ESCET and ESCET are trademarks of Eclipse Foundation, Inc.

[8] Wan J Fokkink, Martijn A Goorden, Dennis Hendriks, DA van Beek, Albert T Hofkamp, Ferdie FH Reijnen, LFP Etman, Lars Moormann, Joanna M van de Mortel-Fronczak, Michel A Reniers, et al. Eclipse ESCET™: the eclipse supervisory control engineering toolkit. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 44–52. Springer, 2023.

[9] Dennis Hendriks, Michel Reniers, Wan Fokkink, and Wytse Oortwijn. Overview and Performance Evaluation of Supervisory Controller Synthesis with Eclipse ESCET v4.0, 2025.

[10] Dirk A van Beek, Wan J Fokkink, Dennis Hendriks, Albert Hofkamp, Jasen Markovski, Joanna M van de Mortel-Fronczak, and Michel A Reniers. CIF 3: Model-based engineering of supervisory controllers. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 575–580. Springer, 2014.

[11] Amir Pnueli. The temporal logic of programs. In *18th annual symposium on foundations of computer science (sfcs 1977)*, pages 46–57. ieee, 1977.

[12] Edmund M Clarke, E Allen Emerson, and A Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[13] Max H de Queiroz, José ER Cury, and Walter Murray Wonham. Multitasking supervisory control of discrete-event systems. *Discrete Event Dynamic Systems*, 15(4):375–395, 2005.

[14] W.M. Wonham, Kai Cai, and Karen Rudie. Supervisory control of discrete-event systems: A brief history. *Annual Reviews in Control*, 45:250–256, 2018.

[15] Christos G Cassandras and Stephane Lafortune. *Introduction to discrete event systems*. Springer, 2007.

[16] Lucien Ouedraogo, Ratnesh Kumar, Robi Malik, and Knut Akesson. Nonblocking and Safe Control of Discrete-Event Systems Modeled as Extended Finite Automata. *IEEE Transactions on Automation Science and Engineering*, 8(3):560–569, 2011.

[17] Yde Venema. Lectures on the modal μ-calculus. *Renmin University in Beijing (China)*, 2008.

[18] Sjoerd Cranen, Jan Friso Groote, Jeroen JA Keiren, Frank PM Stappers, Erik P De Vink, Wieger Wesselink, and Tim AC Willemse. An overview of the mCRL2 toolset and its recent advances. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 199–213. Springer, 2013.

[19] Markus Skoldstam, Knut Akesson, and Martin Fabian. Modeling of discrete event systems using finite automata with variables. In *2007 46th IEEE Conference on Decision and Control*, pages 3387–3392. IEEE, 2007.

[20] Zhennan Fei, Sajed Miremadi, Knut Åkesson, and Bengt Lennartson. Efficient symbolic supervisor synthesis for extended finite automata. *IEEE Transactions on Control Systems Technology*, 22(6):2368–2375, 2014.

[21] Stéphane Lafortune. Discrete event systems: Modeling, observation, and control. *Annual Review of Control, Robotics, and Autonomous Systems*, 2(1):141–159, 2019.

[22] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[23] Gianfranco Ciardo, Robert Marmorstein, and Radu Siminiceanu. The saturation algorithm for symbolic state-space exploration. *International Journal on Software Tools for Technology Transfer*, 8(1):4–25, 2006.

[24] Jeroen JA Keiren and Michel A Reniers. Overview of Controllability Definitions in Supervisory Control Theory. *arXiv preprint arXiv:2508.05177*, 2025.

[25] Aida Rashidinejad, Michel Reniers, and Martin Fabian. Supervisory control synthesis of timed automata using forcible events. *IEEE Transactions on Automatic Control*, 69(2):1074–1080, 2023.

[26] Saul A Kripke. A completeness theorem in modal logic1. *The journal of symbolic logic*, 24(1):1–14, 1959.

[27] Edmund M Clarke, Thomas A Henzinger, Helmut Veith, Roderick Bloem, et al. *Handbook of model checking*, volume 10. Springer, 2018.

[28] Wan Fokkink and Martijn Goorden. Offline supervisory control synthesis: taxonomy and recent developments. *Discrete Event Dynamic Systems*, 34(4):605–657, 2024.

[29] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 179–190, 1989.

[30] Zahra Ramezani, Jonas Krook, Zhennan Fei, Martin Fabian, and Knut Akesson. Comparative case studies of reactive synthesis and supervisory control. In *2019 18th European Control Conference (ECC)*, pages 1752–1759. IEEE, 2019.

[31] AC Van Hulst, Michel A Reniers, and Wan J Fokkink. Maximal synthesis for Hennessy-Milner logic. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(1):1–21, 2015.

[32] AC Van Hulst, Michel A Reniers, and Wan J Fokkink. Maximally permissive controlled system synthesis for non-determinism and modal logic. *Discrete Event Dynamic Systems*, 27(1):109–142, 2017.

[33] Blake C Rawlings, Stéphane Lafortune, and B Erik Ydstie. Supervisory control of labeled transition systems subject to multiple reachability requirements via symbolic model checking. *IEEE Transactions on Control Systems Technology*, 28(2):644–652, 2018.

[34] Sahar Mohajerani, Robi Malik, Andrew Wintenberg, Stéphane Lafortune, and Necmiye Ozay. Divergent stutter bisimulation abstraction for controller synthesis with linear temporal logic specifications. *Automatica*, 130:109723, 2021.

[35] Kiam Tian Seow. Supervisory control of fair discrete-event systems: A canonical temporal logic foundation. *IEEE Transactions on Automatic Control*, 66(11):5269–5282, 2020.

[36] Ryohei Oura, Toshimitsu Ushio, and Ami Sakakibara. Bounded synthesis and reinforcement learning of supervisors for stochastic discrete event systems with LTL specifications. *IEEE Transactions on Automatic Control*, 69(10):6668–6683, 2024.

[37] Anatoli A Tziola. *Task Planning and Control Synthesis for Multi-Agent Systems*. PhD thesis, 2024.

[38] Bernd Finkbeiner, Niklas Metzger, Satya Prakash Nayak, and Anne-Kathrin Schmuck. Synthesis of Universal Safety Controllers. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 177–197. Springer, 2025.

[39] Kittiphon Phalakarn, Sasinee Pruekprasert, and Ichiro Hasuo. Winning Strategy Templates for Stochastic Parity Games Towards. In *Theoretical Aspects of Computing–ICTAC 2024: 21st International Colloquium, Bangkok, Thailand, November 25–29, 2024, Proceedings*, volume 15373, page 197. Springer Nature, 2024.

[40] Rômulo Meira-Góes, Ian Dardik, Eunsuk Kang, Stéphane Lafortune, and Stavros Tripakis. Safe environmental envelopes of discrete systems. In *International Conference on Computer Aided Verification*, pages 326–350. Springer, 2023.

[41] Milad Kazemi Mehrabadi. *Data-driven approaches for formal synthesis of cyber-physical systems*. PhD thesis, Newcastle University, 2023.

[42] Kiam Tian Seow. Decentralized Supervisory Control of Discrete-Event Systems in Canonical Temporal Logic. *IEEE Access*, 2025.

[43] Bruno Lacerda. Supervision of discrete event systems based on temporal logic specifications. *Instituto Superior Técnico, Universidade Tecnica de Lisboa, Lisbon, Portugal, Doctor of Philosophy (Ph. D) Thesis*, 2013.

[44] Sander Thuijsman, Dennis Hendriks, Rolf Theunissen, Michel Reniers, and Ramon Schiffelers. Computational effort of BDD-based supervisor synthesis of extended finite automata. In *2019 IEEE 15th International Conference on Automation Science and Engineering (CASE)*, pages 486–493. IEEE, 2019.

[45] Robin Milner. *Communication and concurrency.* Prentice-Hall, Inc., 1989.

[46] Michel Reniers and Calvin Dingemans. Supervisory control with absent-state explanations for coloured finite automata. *IFAC-PapersOnLine*, 55(28):173–179, 2022.

[47] Tim Korssen, Victor Dolk, Joanna M. van de Mortel-Fronczak, Michel A. Reniers, and Maurice Heemels. Systematic Model-Based Design and Implementation of Supervisors for Advanced Driver Assistance Systems. *IEEE Transactions on Intelligent Transportation Systems*, 19(2):533–544, 2018.

[48] W Murray Wonham and Kai Cai. *Supervisory Control of Discrete-Event Systems.* Springer, 2019.

[49] Maurice H. ter Beek, Michel A. Reniers, and Erik P. de Vink. Supervisory Controller Synthesis for Product Lines Using CIF 3. *Proc. 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques (ISoLA), Part I*, 9952:856–873, 2016.

[50] Sander B. Thuijsman and Michel A. Reniers. Supervisory Control for Dynamic Feature Configuration in Product Lines. *ACM Transactions on Embedded Computing Systems*, 2023.

[51] F. F. H. Reijnen, M. A. Reniers, J. M. van de Mortel-Fronczak, and J. E. Rooda. Structured Synthesis of Fault-Tolerant Supervisory Controllers. *Proc. 10th Symposium on Fault Detection, Supervision and Safety for Technical Processes (SAFEPROCESS), IFAC-PapersOnLine*, 51(24):894–901, 2018.

[52] Rong Su, Jan H. van Schuppen, and Jacobus E. Rooda. Aggregative Synthesis of Distributed Supervisors Based on Automaton Abstraction. *IEEE Transactions on Automatic Control*, 55(7):1627–1640, 2010.

[53] F. F. H. Reijnen, M. A. Goorden, J. M. van de Mortel-Fronczak, M. A. Reniers, and J. E. Rooda. Application of Dependency Structure Matrices and Multilevel Synthesis to a Production Line. *Proc. 2nd Conference on Control Technology and Applications (CCTA)*, pages 458–464, 2018.

[54] R. J. M. Theunissen, M. Petreczky, R. R. H. Schiffelers, D. A. van Beek, and J. E. Rooda. Application of Supervisory Control Synthesis to a Patient Support Table of a Magnetic Resonance Imaging Scanner. *IEEE Transactions on Automation Science and Engineering (TASE)*, 11(1):20–32, 2014.

[55] Rolf J. M. Theunissen. *Supervisory Control in Health Care Systems.* PhD thesis, Eindhoven University of Technology, The Netherlands, 2015.

[56] Lei Feng, Kai Cai, and W. M. Wonham. A structural approach to the non-blocking supervisory control of discrete-event systems. *The International Journal of Advanced Manufacturing Technology*, 41:1152–1168, 2009.

[57] Stefan T. J. Forschelen, Joanna M. van de Mortel-Fronczak, Rong Su, and Jacobus E. Rooda. Application of supervisory control theory to theme park vehicles. *Discrete Event Dynamic Systems*, 22:511–540, 2012.

[58] L. J. van der Sanden, Michel A. Reniers, Marc C. W. Geilen, A. A. Basten, Johan Jacobs, Jeroen P. M. Voeten, and Ramon R. H. Schiffelers. Modular Model-Based Supervisory Controller Design for Wafer Logistics in Lithography Machines. *Proc. 18th Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 416–425, 2015.

[59] F. F. H. Reijnen, M. A. Goorden, J. M. van de Mortel-Fronczak, and J. E. Rooda. Supervisory control synthesis for a waterway lock. *Proc. 1st Conference on Control Technology and Applications (CCTA)*, pages 1562–1563, 2017.

[60] Terence Beijloos. Scripts and models to reproduce experiments. `https://doi.org/10.5281/zenodo.17856810`, December 2025.

[61] Michel Reniers and Jeroen JA Keiren. Validation of supervisory control synthesis tool CIF using model checker mCRL2. In *2024 IEEE 20th International Conference on Automation Science and Engineering (CASE)*, pages 1437–1442. IEEE, 2024.