Tesis Doctoral

Síntesis dirigida de controladores para sistemas de eventos discretos

Ciolek, Daniel Alfredo

2018

UNIVERSIDAD DE BUENOS AIRES

Facultad de Ciencias Exactas y Naturales

Departamento de Computación

Síntesis dirigida de controladores para sistemas de eventos discretos

Tesis presentada para optar al título de Doctor de la
Universidad de Buenos Aires en el área de Ciencias de la Computación

Lic. Daniel Alfredo Ciolek

Director de Tesis: Dr. Sebastián Uchitel
Consejero de Estudios: Dr. Diego Garbervetsky
Lugar de Trabajo: Laboratorio de Fundamentos y Herramientas para la Ingeniería del Software (LaFHIS), FCEyN, UBA

Buenos Aires, 14 de Agosto de 2018

Fecha de defensa: 12 de Septiembre de 2018

Directed controller synthesis for discrete event systems Abstract

The problem of automatically constructing a software component such that when executed in a given environment satisfies a goal is recurrent in software engineering and, in particular, in the field of discrete event systems. Supervisory control, reactive synthesis and automated planning are three disciplines which fit into this vision. Arising from different communities, they consider distinct perspectives on representational and computational aspects. Interestingly, the three disciplines share the important characteristic that their problems' semantics are based on sorts of transitions systems, which are often exponential with respect to the size of compact input specifications. In this thesis we study the synthesis problem from the perspective of supervisory control highlighting the relations between these three disciplines.

We start by showing how reactive synthesis and automated planning can be leveraged effectively to solve supervisory control problems of deterministic discrete event systems. To do so, we propose efficient translations of the supervisory control problem into the reactive synthesis and automated planning frameworks. Notably, our translations capture the compositional and reactive nature of control specifications, avoiding a potential exponential explosion found in similar approaches. We report on an experimental evaluation comparing the efficacy of different tools from the three disciplines. The results show that our translations allow to transparently apply techniques from reactive synthesis and automated planning with an efficiency that rivals that of native supervisory control tools.

We continue presenting the directed controller synthesis technique for discrete event systems. Inspired by the combination of techniques, this method explores the solu-

tion space for supervisors guided by a domain-independent heuristic. The heuristic is derived from an abstraction based on the componentized way in which complex environments are described. The abstraction can be seen as a relaxed version of the problem, whose simpler solution can provide insights on how to solve the original problem. We propose two heuristics, the first is extracted from an abstraction built by considering a simplified form of composition, while the second attempts to discover dependencies between the intervening components. Then, by building the composition of the components on-the-fly, we obtain a solution exploring only a reduced portion of the state space. We report on an evaluation of the technique comparing it to well-known approaches from the three disciplines and show that our method performs well even as the size of the state space grows.

Finally, we discuss an extension to the directed controller synthesis technique that allows the computation of supervisors under partially observable and non-deterministic environments, which incurs in added complexity. We exploit the link between partially observable control and non-determinism and show that we can reduce the former into the latter compositionally. Additionally, we point out that in this setting the existence of a solution may depend on the interaction model between the controller-to-be and the environment, and show how our technique adapts to two relevant interaction models.

Keywords: Discrete Event Systems, Supervisory Control, Reactive Synthesis, Automated Planning, Non-deterministic and Partially Observable Environments.

Síntesisi dirigida de controladores para sitemas de eventos discretos

Resumen

El problema de construir automáticamente un componente de software que al ser ejecutado en un ambiente dado satisfaga un objetivo, es recurrente en la ingeniería del software y en particular en el campo de los sistemas de eventos discretos. El control supervisor, la síntesis de sistemas reactivos y la planificación automática son tres disciplinas que se alinean con esta visión. Proviniendo de distintas comunidades, consideran distintas perspectivas con respecto a aspectos de representación y cómputo. Resulta interesante que las tres disciplinas comparten la característica importante de que la semántica de sus problemas está basada en variantes de sistemas de transiciones, que frecuentemente resultan ser exponenciales con respecto al tamaño de especificaciones compactas. En esta tesis estudiamos el problema de síntesis desde la perspectiva del control supervisor resaltando la relación entre las tres disciplinas.

Comenzamos mostrando cómo la síntesis reactiva y la planificación automática pueden ser utilizadas efectivamente para resolver problemas de control supervisor de sistemas de eventos discretos determinísticos. Para lograrlo, proponemos traducciones eficientes del problema de control supervisor en el marco de la síntesis reactiva y la planificación automática. Notablemente, nuestras traducciones capturan la naturaleza composicional y reactiva de las especificaciones de control, evitando la explosión exponencial a la que están sujetos acercamientos similares. Reportamos los resultados de una evaluación experimental comparando la eficacia de distintas herramientas provenientes de las tres disciplinas. Los resultados muestran que nuestras traducciones permiten aplicar transparentemente técnicas de síntesis reactiva y

planificación automática con una eficiencia competitiva con las herramientas nativas al control supervisor.

Continuamos presentando una técnica de síntesis dirigida de controladores para sistemas de eventos discretos. Inspirado en la combinación de técnicas, este método explora el espacio de solución buscando supervisores guiado por una heurística independiente del dominio. La heurística es derivada de una abstracción basada en la forma componetizada en la que se describen ambientes complejos. La abstracción puede verse como una versión relajada del problema, cuya solución más simple puede proveer indicios sobre cómo resolver el problema original. Luego, construyendo la composición de los componentes "sobre la marcha" obtenemos una solución explorando sólo una porción reducida del espacio de estados. Presentamos una evaluación de la técnica comparándola con acercamientos establecidos de las tres disciplinas y mostramos que nuestro método se desempeña bien incluso a medida que el espacio de estados crece.

Finalmente, discutimos una extensión a la síntesis dirigida de controladores que permite computar supervisores en ambientes parcialmente observables y no-determinísticos, incurriendo en una complejidad adicional. Aprovechamos el vínculo entre observabilidad parcial y no-determinismo y mostramos que podemos reducir el primer problema en el segundo composicionalmente. Adicionalmente, hacemos notar que en este contexto la existencia de una solución puede depender del modelo de interacción entre el controlador a sintetizar y el ambiente, y mostramos cómo nuestra técnica se adapta a dos modelos de interacción relevantes.

Palabras clave: Sistemas de Eventos Discretos, Control Supervisor, Síntesis de sistemas Reactivos, Planificación Automática, Entornos No-determinísticos y Parcialmente Observables.

Agradecimientos

Estas páginas no podrían existir sin la ayuda y el apoyo de muchas personas que formaron parte de mi vida mientras llevé a cabo este trabajo.

Antes que nada quiero agradecerle a Sebastián Uchitel, mi director, por su guía y dedicación. Sebastián tuvo la altura de trabajar con mi terquedad, dándome libertad y ayudándome a superar los diversos obstáculos que se nos fueron presentando.

Quiero extender este agradecimiento a todos con los que tuve la suerte de colaborar para lograr los resultados de esta tesis. Victor Braberman, que con su paciencia y aguda observación de los detalles, ha sabido corregir y mejorar los resultados de este trabajo. Nicolás D'Ippolito, que con su humor y complicidad siempre logró alivianar el trajín diario, y generar nuevas posibilidades. Nir Piterman, que con su generosidad y excepcionalidad me ha enseñado a tachar y reescribir las veces que sea necesario para lograr el resultado deseado. Sebastian Sardiña, con una pasión que contagia energía y una hospitalidad que logró que más un colega lo sienta familia.

También quiero agradecer a todos los miembros de LaFHIS y PPL por la buena onda, las charlas, los cafés, los ánimos y el compañerismo. En estos años fuimos muchos los que compartimos este espacio, por lo que también pido disculpas si en lo que sigue olvido mencionar a alguno, todos son importantes: Hernán, Die, JP, Tebi, Ferto, Negro, Guido, Gerva, Rodri, Zoppirata, Capuchari, Turco, Eze, Fernán, Marian, Dani$^{\times 2}$, Nati$^{\times 2}$, Chris, Iván$^{\times 2}$, Alexis, Vir, Manu, Marion, Billy, Juli y Fabi.

Por último quiero agradecer a mis padres, María del Cármen y Ricardo, por su afecto y apoyo incondicional. También a mis suegros, cuñados y sobrinos que han sabido llenarme de alegría, Alicia, Tony, Andrea, Diego, Rami, Cata, Paula, Guille, Rochi, Luqui, Diego y Alejandra. Y sobretodo le agradezco a mi mujer por su amor, comprensión y compañía; Natt te amo.

Contents

List of Figures

List of Tables

Listings

1

Introduction

1.1 Motivation

In the last decades, the evolution of computing technologies has made automated dynamic systems ubiquitous in our every day life. Examples are all around us: computer and communication networks; automated manufacturing; air traffic control; business management processes; and so forth. As the complexity of these systems increases, the likelihood of encountering errors also grows. Such errors may have catastrophic consequences, not only in monetary terms, but also on human life. Therefore, there is a growing interest in methodologies that can prove the correctness of these critical systems. It is expected that these methodologies will result in improved quality as well as in reduced development costs.

The traditional engineering approach to the construction of critical systems relies on formal verification (in contrast with validation/testing approaches usually used for non-critical systems [1]). First, requirements are expressed in terms of the phenomena on the interface between the machine we are to build and the world in which the problem is to be solved, together with assumptions and goals of the system to be [2]. Second, formal models are built,

which can be studied and modified until confidence in their adequacy with respect to the requirements is obtained. The advantage of models is that they are easier to reason about, and their construction is simpler compared to the system itself. Third, the actual system is developed, usually following good practices to reduce the chance and impact of human errors. And fourth, the implementation is formally verified against the model, using techniques such as model checking. Of course the process is not linear but usually progresses in a spiraled fashion iterating as requirements are included.

An alternative approach, the one studied in this thesis, is the automatic synthesis of a system from its requirements specification. This approach attempts to automatically process formal models in order to obtain a correct by construction system implementation. In its full generality, this problem cannot be algorithmically solved, and hence we work within a framework able to express a restricted set of system requirements. Similarly to model checking, we work with a finite model of a system, which can be expressed in terms of communicating state machines. Alas, model checking and synthesis share an inherent disadvantage known as the *state explosion problem*. This problem describes the exponential relation of the number of states in the model to the number of intervening components in the system, and represents a barrier to the applicability of the techniques.

1.2 Discrete Event Systems

Discrete Event Systems (DES) are discrete-state, event-driven dynamic systems that react to the occurrence of events. DES arise when the state space of a system is naturally described by a discrete set, and state transitions (associated with events) are only observed at discrete points in time (i.e., they occur instantaneously). An event may be identified with some controllable action, or with some uncontrollable occurrence dictated by nature which may or may not be observable through sensors. Despite their simplicity DES capture a wide range of practical systems.

DES are usually modeled through finite state transition systems such as automata and Petri-nets. These formalisms have in common the fact that they represent languages, and differ by how they represent state information. Im-

portantly, both formalisms are amenable to composition operations, which allows building the model of a system from models of intervening components. This is very convenient for modeling, plus it also allows to tackle analysis and synthesis by making use of structural properties of the model.

The dynamics of DES are ruled by state transitions that occur at certain points in time through instantaneous events. The *execution model* can be asynchronous, that is, an event may occur at various points in time not necessarily known in advance; or synchronous, which assumes the existence of a clock, and in which events only occur at clock ticks. Furthermore, the *communication model* can in turn be asynchronous, in which events at each intervening component may occur at any point, even simultaneously; or synchronous, in which for an event to occur, all the intervening components have to "synchronize" on its occurrence (i.e., effectively communicating through the event). These distinctions give rise to the terms time-driven and event-driven systems [3]. Despite their differences, algorithmic solutions developed for one may be applicable to the other with only small modifications.

As the complexity of DES grows the state explosion problem makes their analysis intractable. Thus, there is a growing interest in developing techniques capable of coping with the state blowup. From a theoretical point of view worst case complexity may not improve, yet algorithms targeting applicability exist and provide huge benefits in practice.

1.3 The Synthesis Problem

The *synthesis problem* deals with the automatic construction of a system from a formal specification, as an alternative to manual development and verification. Synthesis aims at generating an operational component from a declarative input specification, such that it guarantees by construction to achieve the system requirements.

Supervisory Control [4, 3], Reactive Synthesis [5, 6] and Automated Planning [7, 8] are three disciplines that tackle the synthesis problem. Arising from different communities, they consider distinct perspectives on representational and computational aspects. Interestingly, the three disciplines share

the important characteristic that input specifications are given via *compact descriptions*. That is, the problems' semantics are based on sorts of transitions systems, which are often exponential with respect to the size of these compact descriptions.

In supervisory control, DES are expressed compactly by relying on a modular approach based on the *parallel composition* of multiple interacting components [9], referred to as the *plant*. Supervisory control aims at controlling DES to achieve certain guarantees, this is done by deploying a so-called "supervisor" that dynamically disables controllable events while monitoring uncontrollable events. Traditional supervisory control techniques look for maximally permissive supervisors, which has been argued to require a prohibitive computational cost [10]. Directors (i.e., minimally permissive supervisors) [11] have been proposed as a computational efficient alternative, still effective synthesis procedures have not yet been developed for this particular case.

In the field of reactive synthesis the specification is usually expressed compactly in a temporal logic [6], such as Computational Tree Logic (CTL) or Linear Temporal Logic (LTL). Controllers are seen as open dynamical systems, in the sense that they have input and output signals which affect the system dynamics. The specification of a reactive synthesis problem describes the desired behavior of the controller on the interface level (i.e., the values of signals). In this setting, complex systems are usually specified in a modular fashion as the conjunction of simpler sub-modules. This approach has been mainly used for hardware synthesis [12], but recently there have also been attempts to apply it to software [13].

Automated Planning (or planning for short) stems from a very different tradition, namely, Knowledge Representation within Artificial Intelligence. A planning problem is specified by describing the preconditions and effects of actions, together with the goal to be achieved. Based on reasoning about action languages, powerful techniques have been developed over the years [14]. However, the work in planning, unlike that in supervisory control and reactive synthesis, has been oriented mainly towards non-reactive environments. Still, advances in planning have led to tackle non-deterministic problems which can be used to model reactivity [15].

Interestingly, the different representations can be interpreted in terms of finite state transition systems. In spite of considering problem definitions with subtle variations, the three areas can be seen as looking for a system subset satisfying a given property. Usually in planning, the goal is to reach a particular state, that is, a reachability objective. Instead, in reactive synthesis, a liveness requirement such as a Büchi acceptance condition [6], is pursued. Whereas, in supervisory control we look for a subset of the system that can always continue to be productive.

Despite differences in representational aspects, the three disciplines share common problems and objectives. In particular, the state explosion has been dealt with in the three fields relying on different approaches. For instance, in supervisory control compositional analyses (e.g. [16]) are performed on individual components allowing for a (potentially) efficient merging procedure that meets the problem requirements; in reactive synthesis symbolic representations (e.g. [17]) are used to encode the problem in a "compressed" structure which can be processed efficiently; and in automated planning informed search procedures (e.g. [18]) are used to find a solution by exploring (hopefully) only a reduced portion of the state space.

The synthesis problem is known to be computationally challenging. Supervisory control is PTIME with respect to the size of the overall plant [9], but since we consider a compositional description of the plant, the problem is ultimately EXPTIME with respect to the size of the individual components. Reactive synthesis is in worst case 2EXPTIME-complete [6], yet by restricting the input (e.g., relying on CTL or GR(1) requirements [19, 20]) the second exponential explosion can often be circumvented. Similarly, classical deterministic planning is PSPACE-complete with respect to the succinct planning specification, the addition of non-deterministic actions makes the problem EXPTIME-complete [21].

Motivated by their similarities, there has been great interest in relating these fields and finding opportunities for cross-fertilization [11, 22, 23, 24, 25, 26, 27]. In this thesis we review techniques from the different fields and, by combining them, we devise a novel algorithmic solution called Directed Controller Synthesis (DCS). The DCS method explores the solution space guided by domain-independent heuristics, which are derived from abstrac-

tions based on the componentized way in which complex environments are described. Then, by building the composition of components on-the-fly, DCS obtains a solution exploring only a reduced portion of the state space.

We devise two abstractions from which heuristic estimates can be efficiently extracted to guide the on-the-fly exploration. The first, inspired in a classical heuristic used in planning [14] and directed model checking [28], relaxes the problem by considering a monotonically increasing set of states. That is, the abstraction behaves as if, after a transition, not only a new state is covered but also the source state is not abandoned. This takes into account the effects of synchronization among components (up to some point) only considering the sum of states of intervening components in worst case. The second, computes a dependencies graph over a subset of transitions, that is, considering just the sum of transitions in worst case. The graph contains causal relations between transitions, providing insights about the order in which transitions need to be taken in order to achieve the goal.

Furthermore, we provide polynomial-time compilations that allow to leverage on the advances on reactive synthesis and planning to solve compositional supervisory control problems. These translations allow us to evaluate the effectiveness of the different techniques over a supervisory control benchmark, and to validate DCS. From this evaluation we determine that the on-the-fly exploration approach is competitive with state-of-the-art techniques.

1.4 Goals and Contributions

In this work, we contribute to recent efforts in relating the fields of supervisory control, reactive synthesis and automated planning [11, 22, 23, 24, 25, 26, 27]. Furthermore, we propose new algorithmic solutions based on the combination of techniques from these fields.

The main contributions of this thesis can be summarized as follows:

1. A novel algorithmic approach (DCS) to solve the compositional supervisory control problem, inspired by the combination of techniques from the fields of supervisory control, reactive synthesis and automated planning. The results extracted from our evaluation show that DCS is able to

cope with problems orders of magnitude larger than existing monolithic and compositional approaches for supervisory control.

2. A polynomial-time compilation from supervisory control, which allows to solve deterministic compositional control problems by leveraging the advances in:
   (a) reactive synthesis; and
   (b) non-deterministic planning.

Some additional contributions of this thesis can be summarized as:

3. The development of a benchmark of problems of interest to multiple fields of study, which thanks to the use of a declarative specification language can be easily scaled to different sizes.
4. The identification of different interaction models arising in partially observable and non-deterministic environments; the analysis of their impact on realizability; and the extension of DCS to consider these settings.

1.5 Document Organization

This document is organized as follows. In Chapter 2 we start by providing required formal background. In Chapter 3 we present distinctive case studies that conform a benchmark, which we use for motivation and evaluation. In Chapter 4 we present the reduction from compositional supervisory control problems into reactive synthesis and automated planning, and validate the applicability of the translation against the proposed benchmark. In Chapter 5 we present the DCS method and report on an evaluation by comparing its performance on the benchmark. In Chapter 6 we discuss extensions to the DCS method that allows it to tackle partially observable and non-deterministic problems, under two relevant interaction models. Finally, in Chapter 7 we document the conclusions of this thesis together with open questions and avenues for future work.

1.6 Resumen del capítulo 1

En las últimas décadas, la evolución de las tecnologías de la información a generalizado el despliegue de sistemas dinámicos automáticos en muchos aspectos de la vida cotidiana. A medida que la complejidad de estos sistemas aumenta, la probabilidad de encontrar errores crece. Tales errores pueden tener consecuencias catastróficas, no sólo en términos monetarios sino también en vidas humanas. Por lo tanto, existe un interés creciente en metodologías que prueben la correctitud de estos sistemas críticos. Se espera que estas metodologías resulten en una mejora de calidad como también en una reducción de los costos de desarrollo.

El acercamiento tradicional a la construcción de sistemas críticos se basa en la verificación formal (en contraste con los acercamientos de validación/testing normalmente utilizados para sistemas no-críticos). Este acercamiento requiere primero expresar los requerimientos, las hipótesis y los objetivos del sistema con modelos formales; para posteriormente efectuar un proceso de verificación usando técnicas como las del model-checking. La ventaja de los modelos es que resultan más fácil de construir y analizar en comparación con la implementación del sistema.

Un acercamiento alternativo, el estudiado en esta tesis, es la síntesis automática de un sistema a partir de la especificación de sus requerimientos. Desafortunadamente, en su versión más general este problema no puede resolverse algoritmicamente. En su lugar, trabajamos en un marco que permite expresar un conjunto restringido de requerimientos. Al igual que con model-checking, la síntesis se ve sujeta a una limitación inherente conocida como el *problema de explsión del espacio de estados*. Este problema describe la relación exponencial entre el número de estados en el modelo y el número de componentes intervinientes en el sistema, y representa la principal barrera a la aplicabilidad de las técnicas.

El Control Supervisor [4, 3], la Síntesis de sistemas Reactivos [5, 6] y la Planificación Automática [7, 8] son tres disciplinas que abordan el problema de síntesis. Proviniendo de diferentes comunidades, consideran distintas perspectivas en aspectos de representación y cómputo. No obstante, las tres disciplinas comparten la importante característica que admiten especificaciones

dadas mediante *descripciones compactas*. Es decir, las semánticas de los problemas están basadas en sistemas de transiciones que frecuentemente resultan exponenciales con respecto al tamaño de estas descripciones compactas.

A pesar de sus diferencias, las tres disciplinas comparten problemas y objetivos comunes. Motivado por sus similitudes, se ha despertado un gran interés en relacionar estos campos de estudio en pos de encontrar oportunidades de retroalimentación [11, 22, 23, 24, 25, 26, 27]. En esta tesis revisamos técnicas de los tres campos y, combinándolas, diseñamos una solución algorítmica novedosa llamada la Síntesis Dirigida de Controladores (DCS por sus siglas en ingles). El método DCS explora el espacio de solución guiado por una heurística independiente del dominio, que se deriva de abstracciones basadas en la forma componetizada en la que se describen ambientes complejos.

Además, presentamos traducciones que, ejecutando en tiempo polinomial, permiten aprovechar los avances en síntesis reactiva y planificación para resolver problemas de control supervisor. Estas traducciones nos permiten evaluar la efectividad de las distintas técnicas para resolver problemas de control supervisor, y para validar DCS. De la evaluación se desprende que los métodos de exploración dirigidos son competitivos con las otras técnicas del estado del arte.

2

Preliminaries

2.1 Translation based approaches

A common approach to relate different fields tackling similar problems is to provide translations between their problem formalizations. Thus, translations between supervisory control, reactive synthesis and automated planning have been proposed in the literature. However, to the best of our knowledge none of the existing approaches attempt to reduce a supervisory control problem into planning or reactive synthesis *exploiting the compositional aspects of supervisory control specifications*.

Modular descriptions are central to supervisory control, and hence in this thesis we contribute to relating the different fields by providing a translation that takes composition into account. That is, our compilation avoids the construction of a potentially exponential semantic model by explicitly capturing the effects of parallel composition and synchronization among components in polynomial time. In this section we discuss approaches that share some characteristics with our work.

In [11] and [22] reductions from supervisory control to reactive synthesis are presented. The approaches are *monolithic*, in the sense that they consider

as input to the supervisory control problem a single automaton (the result of the composition of multiple interacting components). Thus, despite being formally correct, these approaches are not practical for handling complex compositional specifications since computing the composition may result in an exponential blowup.

In [27] a reduction from planning to supervisory control is presented. However, the work focuses on explicitly characterizing fairness assumptions. The proposed reduction is *monolithic*, and hence it can also incur in an exponential cost since it requires to completely construct the underlying semantic model.

In [23] a compilation from reactive synthesis problems into planning is presented. Interestingly, the compilation passes through an intermediate representation based on a Finite State Machine (FSM), which resembles the formalism used in supervisory control. Similarly, in [26] a compilation from reactive synthesis into planning that passes through a non-deterministic Büchi automaton (instead of an FSM), is presented together with a number of optimizations. However, both approaches rely on *monolithic* FSM/automata and hence do not tackle the challenges of the compositional case.

In [29] a reduction from ConGolog (a logical programming language for concurrent agents) to situation calculus (a common logical interpretation for planning) is presented. This translation shares some characteristics with ours, namely the schematization of different phases in the execution semantics of concurrent models. However, their translation relies on building a Petri-Net, while we explicitly encode the synchronization mechanism. That is, the Petri-Net accommodates a compositional approach by relying in semantics for multiple tokens, while we directly encode the semantics of parallel composition.

In [30] a framework for the composition of "behaviors" is presented. The technique allows to compute a supremal controller from smaller controllers for sub-modules, such that it remains correct even under the presence of exogenous events. Despite taking as input a compositional specification, the synthesis approach is *monolithic* (i.e., it works over the composition of behaviors). We believe our translations can prove useful at extending said work

by providing insights into how to exploit the compositional nature of the problem.

The existing body of work highlights the similarities between the disciplines and the interest of the different communities in relating the fields. Despite the fact that traditional supervisory control takes a *monolithic* approach to the synthesis of a maximally permissive supervisor, there is a growing interest in *compositional* [31, 32] and *non-maximal* approaches [10, 33]. The combination of the two brings supervisory control close to planning and reactive synthesis, making the question on their relationship relevant. Thus, in this thesis we build in this direction by providing provably correct translations from non-maximal compositional supervisory control problems that do not incur in an exponential blowup.

2.2 Software Tools

Numerous software tools have been developed to solve the synthesis problem. Traditional synthesis techniques can be classified in various ways. In Table 2.1 we show a classification of some publicly available software tools. In our evaluation we consider the subset of the best performing tools from this list.

Tool	Input	Compositional	Symbolic	On-the-fly	Field
DCS	FSP	✓	✗	✓	Supervisory Control
MTSA	FSP	✗	✗	✗	
SUPREMICA	AUT	✓	✗	✗	
MBP	CTL	✗	✓	✗	Reactive Synthesis
PARTY	CTL*	✗	✓	✗	
SLUGS	LTL	✗	✓	✗	
GPT	PDDL	✗	✗	✓	Automated Planning
MYND	PDDL	✗	✗	✓	
PRP	PDDL	✗	✗	✓	

Table 2.1. Tools classification.

For all these tools, input specifications share the common characteristic that are *compact*, in the sense that the actual semantic models they represent are usually exponential with respect to the size of these descriptions.

Supervisory control tools like MTSA[1] [34] and SUPREMICA [35] take automata with marked states as input. Automata based descriptions are modular since interacting components can be described separately and later combined through parallel composition. In the case of MTSA automata are described in a high level language called Finite State Process (FSP). FSP is a process calculus inspired by CSP [36] designed to be easily machine and human readable. In this thesis we describe control problems using FSP scripts since they allow us to create declarative and scalable problem specifications (for a detailed description of FSP see Appendix A). Remarkably, these modern supervisory control tools, make a trade-off between efficiency and maximality, and do not guarantee maximal solutions (SUPREMICA allows to pursue maximal solutions *optionally* at an additional cost).

Reactive synthesis tools like MBP [37], PARTY [38] and SLUGS [39] take temporal logic formulas as input. Research in reactive synthesis has led to the characterization of formulas, which allow to apply efficient ad-hoc algorithmic solutions, such as CTL [40] and GR(1) [20]. Often practical problems do not require the full generality of temporal logic, and hence the efficient algorithmic solutions for specific cases usually contribute to cope with the state explosion problem.

Planning tools such as GPT [41], MYND [42] and PRP [43] take an input written in the Planning Domain Definition Language (PDDL) [44], which is the de-facto standard for expressing planning problems. In PDDL, problems are expressed in a factorized fashion using actions with preconditions and effects, together with an initial state and a goal. Planing languages, such as PDDL, are grounded in situation calculus [45, 46] a logic formalism designed for representing and reasoning about dynamical domains.

Temporal logics – such as CTL and LTL – have been adopted in both, supervisory control and planning, as a means of describing declarative liveness goals [47, 24, 25]. Some approaches reduce the problems to the reactive synthesis framework, while others explicitly encode the liveness constraints in their own framework.

[1] Algorithms presented in this thesis are included in the MTSA project, which features an open-source collaboration framework: http://mtsa.dc.uba.ar

In terms of representation, monolithic approaches to controller synthesis work with the complete state space but, since it is exponential with respect to the size of the components, explicit representations are impractical. Based on advances in model checking, Binary Decision Diagrams (BDD) [48] have been applied to the synthesis problem to keep a symbolic representation of the state space, in an attempt to cope with the state explosion problem. Contrarily to monolithic approaches, compositional approaches to synthesis analyze each component individually in order to simplify future processing. Individually treated components are usually combined incrementally until a supervisor for the composed system is obtained [9].

In planning, the state space is usually represented explicitly, but the exploration is performed on-the-fly guided by heuristics [14]. Domain-independent heuristics are automatically extracted from problem definitions by analyzing a relaxed version of the problem at hand. Informed search procedures – like Best-First-Search or A* [49] – are used to perform a goal-directed exploration, generally obtaining a solution by inspecting a reduced portion of the state space.

Inspired by planning, informed search procedures have been introduced in model-checking to accelerate the search for an error [28, 50]. Despite the strong relation between model-checking and synthesis along with the positive results obtained by these directed techniques, their application to the synthesis problem has been scarcely explored [51]. In this thesis we present the DCS algorithm, which combines a directed exploration with a compositional analysis of the intervening components. That is, DCS explores the state space on-the-fly directed by an heuristic, which is extracted from an abstraction of the modular input specification.

2.3 Non-Determinism and Partial Observability

In its most general formalization, supervisory control considers a potentially non-deterministic plant. It has been shown that non-deterministic problems can be reduced to the deterministic case [52]. Thus, a common simplifying assumption found in the literature is to restrict the analysis to the determinis-

tic setting. In this thesis we follow this direction, first assuming determinism and then extending our results to the non-deterministic case.

A shortcoming of determinization is that it can potentially incur in an exponential blowup of the state space, and furthermore it is not applicable together with compositional methods. For this reason, we do not rely on a determinization procedure and instead extend our approach to explicitly handle non-determinism. This allows us to take full advantage of the compositional nature of supervisory control descriptions to tackle synthesis efficiently even under non-determinism.

Interestingly, the behavior of a partially observable DES can be captured by means of non-determinism, and hence we show how our technique can be applied in this setting too. Similarly, in reactive synthesis and planning the effects of non-determinism and partial observability are also studied.

A noteworthy related line of work is that of [53], where safety and bounded-liveness properties are used as goals in a deontic input-output automata setting. The technique relies on a standard determinization algorithm, and hence it prevents applying compositional techniques to cope with the state explosion problem.

Remarkably, a compositional approach in non-deterministic environments restricted to safety and co-safety goals is presented in [54]. The technique looks for "most general" strategies for each intervening component, such that when composed the crossed restrictions do not prevent realization of the goals. Since a compositional treatment is performed the technique promises to scale well, yet the requirement for most general strategies imposes additional complexities to the approach.

In [40] CTL* synthesis (considering particular cases for CTL and LTL) is solved considering partial observability by using alternating tree automata. The approach is based on a reduction to satisfiability of a CTL* specification which may produce a non-deterministic supervisor. In [55] a similar approach based on alternating tree automata is presented, but considering a partially observable version of μ-calculus.

In [56] partial observability for planning is considered, and heuristics for directed exploration are devised. Interestingly, a distinction is made, in plan-

ning, between non-determinism with full observability and with partial observability. The former turns out to be equivalent to control under a (potentially adversarial) deterministic environment, while the latter is equivalent to the more general non-deterministic setting.

Summarizing, the synthesis of operational strategies with partially observable and non-deterministic environments is a problem with many facets. In this thesis we consider the synthesis problem under these settings, and compare the approaches taken in the three fields highlighting their differences. Furthermore, we propose an extension to our on-the-fly compositional method avoiding the blowup induced by determinization steps, thus permitting greater scalability.

2.4 Formal Background

In this section we present the relevant notions of the synthesis problem in the context of each field of study. We start by presenting the supervisory control framework, first introduced by Ramadage and Wongham [57]. We continue presenting the reactive synthesis framework introduced by Pnueli and Rosner [6]. And we conclude presenting the automated planning framework extended with non-deterministic effects (required to model reactivity) as described by Rintanen [58]. Given the number of works with extensions and alternatives to these basic concepts, the formalization presented herein may vary slightly with respect to some related literature. When appropriate, we highlight the differences between the formalisms in which we ground our work with other common approaches.

2.4.1 Supervisory Control

Supervisory control for DES focuses on a modular decomposition of control problems with a component interaction model based on events [57]. In particular, control problems for behavior models are expressed as the *parallel composition* (defined broadly as synchronous product) of *deterministic automata*.

Definition 1 (Deterministic Automaton)**.** A *deterministic automaton* is a tuple $T = (S_T, A_T, \rightarrow_T, \bar{t}, M_T)$, where:

- S_T is a *finite set of states*;
- A_T is the automaton *event set*;
- $\rightarrow_T \subseteq (S_T \times A_T \times S_T)$ is a deterministic *transition relation*;
- $\bar{t} \in S_T$ is the *initial state*; and
- $M_T \subseteq S_T$ is a set of *marked states*.

Notation 1 (Steps and Runs)**.** We denote $(t, \ell, t') \in \rightarrow_T$ by $t \xrightarrow{\ell}_T t'$ and call it a *step*. In turn, a *run* on a word $w = \ell_0, \ldots, \ell_k$ in T, is a sequence of steps such that $t_i \xrightarrow{\ell_i}_T t_{i+1}$ for all $0 \leq i \leq k$, denoted by $t_0 \overset{w}{\dashrightarrow}_T t_{k+1}$.

Automata define languages over their event set. Given a set A we denote by A^* the set of finite words of elements of A.

We consider the language *generated* by an automaton T (denoted $\mathcal{L}(T)$) as the set of strings over A_T that follows $\rightarrow_T$. More formally, let $w \in A_T^*$, then $w \in \mathcal{L}(T)$ if and only if there is a run on w in T starting at the initial state $\bar{t}$ and ending in some state $t' \in S_T$, that is $\bar{t} \overset{w}{\dashrightarrow}_T t'$.

As a rule, marked states are used to indicate the termination of a task. Thus, we additionally consider the language *marked* (or *accepted*) by T (denoted $\mathcal{L}_m(T)$) as the set of strings producing runs that end on a marked state. More formally, let $w \in \mathcal{L}(T)$, then $w \in \mathcal{L}_m(T)$ if and only if there is a run on w starting in the initial state $\bar{t}$ and reaching a marked state $t_m \in M_T$, that is $\bar{t} \overset{w}{\dashrightarrow}_T t_m$.

In particular, the language accepted by the parallel composition of the automata intervening in a compositional control problem specifies the desired system behavior.

Definition 2 (Parallel Composition)**.** The *parallel composition* ($\|$) of two automata T and Q is an associative symmetric operator that yields an automaton $T\|Q = (S_T \times S_Q, A_T \cup A_Q, \rightarrow_{T\|Q}, \langle \bar{t}, \bar{q} \rangle, M_T \times M_Q)$, where $\rightarrow_{T\|Q}$ is the smallest relation that satisfies the following rules:

$$\frac{t \xrightarrow{\ell}_T t'}{\langle t,q\rangle \xrightarrow{\ell}_{T\|Q} \langle t',q\rangle}\ell\in A_T\setminus A_Q \qquad \frac{q \xrightarrow{\ell}_Q q'}{\langle t,q\rangle \xrightarrow{\ell}_{T\|Q} \langle t,q'\rangle}\ell\in A_Q\setminus A_T$$

$$\frac{t \xrightarrow{\ell}_T t' \quad q \xrightarrow{\ell}_Q q'}{\langle t,q\rangle \xrightarrow{\ell}_{T\|Q} \langle t',q'\rangle}\ell\in A_T\cap A_Q$$

We highlight three relevant features of the parallel composition. First, the last rule realizes synchronization between components, that is, it enforces the synchronous execution of shared events. Second, the number of states in $T_0\| \ldots \|T_n$ can grow exponentially, as it contains the cross-product of states $(S_{T_0}\times\ldots\times S_{T_n})$. And third, the language accepted by the composition contains the strings that reach marked states in all components *simultaneously*.

A compositional supervisory control problem is defined by two elements, first a set of intervening components and, second, a partition of the event set of the plant in *controllable* and *uncontrollable* events. Given a set of automata and a partition of the event set, we look for a supervisor that can disable some of the controllable events and monitor uncontrollable events; such that every word in the language generated by the *restricted* plant can be extended to a word in the language accepted by the plant. A word w belongs to the language generated by T restricted by a function $\sigma : A_T^* \mapsto 2^{A_T}$ (denoted $\mathcal{L}^\sigma(T)$) if every prefix of w "survives" σ. More formally, let $w = \ell_0, \ldots, \ell_k$ be a word in $\mathcal{L}(T)$, then $w \in \mathcal{L}^\sigma(T)$ if and only if for all $0 \leq i \leq k$:

$$\bar{t} \xrightarrow{\ell_0\ldots\ell_i}_T t_{i+1} \wedge \ell_i \in \sigma(\ell_0, \ldots, \ell_{i-1})$$

Definition 3 (Compositional Supervisory Control Problem)**.** A *Compositional Supervisory Control Problem* is a tuple $\mathcal{E} = (E, A_C)$, where E is a set of automata $\{E_0, \ldots, E_n\}$ (we may abuse notation and also use E to refer to the composition $E_0\| \ldots \|E_n$), and $A_C \subseteq A_E$ is the set of controllable events (i.e., $A_U = A_E \setminus A_C$ is the set of uncontrollable events). A solution for $\mathcal{E}$ is a supervisor $\sigma : A_E^* \mapsto 2^{A_E}$, such that σ is:

- *Controllable*, namely $A_U \subseteq \sigma(w)$ with $w \in A_E^*$; and

- *Non-blocking*, namely for every word $w \in \mathcal{L}^{\sigma}(E)$ there exists a non-empty word $w' \in A_E^*$ such that, the concatenation $ww' \in \mathcal{L}^{\sigma}(E)$ and $\bar{e} \xrightarrow{ww'}_E e_m$ with $e_m \in M_E$.

That is, a supervisor σ is said to be controllable (or admissible) if it only disables controllable events; and it is non-blocking if it is able to restrict the language generated by the plant E to a set of strings that can always be extended to reach a marked state (i.e., a subset of the prefix closure of $\mathcal{L}_m(E)$). Thus, a supervisor is a solution to compositional control problem if, by disabling only controllable events, restricts the plant to states from where marked states are always reachable, but not necessary reached (i.e., uncontrollable events may prevent actually reaching marked states, but may never lead to a deadlock).

Observe that we explicitly define a control problem as taking a compositional description of the plant. We do this since some supervisory control techniques (e.g. [16]) avoid the blowup produced by the parallel composition by reasoning directly on the individual components. Thus, we intentionally prevent the computation of the composition in the problem definition in contrast to traditional monolithic formalizations. Furthermore, monolithic formalizations usually define the supervisory control problem for a plant E with respect to a language $\mathcal{K} \subseteq \mathcal{L}_m(E)$, such that all executions of the plant E are to be restricted within $\mathcal{K}$. This is not necessary in the compositional setting since we can define an auxiliary observer automaton K such that $\mathcal{L}_m(E \| K) = \mathcal{K}$, and rely on the features of the parallel composition to encode the requirement of controlling the plant within $\mathcal{K}$. Moreover, K can in turn be expressed compositionally.

Traditionally, supervisory control techniques look for maximally permissive supervisors. That is, in addition to the base requirements in Definition 3, *maximality* requires that no other "more permissive" supervisor exists (i.e., enabling more controllable events). Maximality comes at a cost in complexity and hence there is a growing interest in less ambitious solution concepts such as directors [10], which require controllers to enable at most one controllable event at each state. Thus, here we follow this trend away from maximality and accept a supervisor satisfying solely the base requirements. Still, in worst

case obtaining even a non-maximal supervisor may incur in a state explosion and require an exponential amount of time.

Example 1. *In Figure 2.1 we show an example of a compositional supervisory control problem where two automata model a manufacturing plant that, upon request, produces one of two products. Automaton C (Figure 2.1.a) represents a customer that can – uncontrollably – request one of two products (r_1 or r_2), and then waits for its delivery (d_1 or d_2 respectively). Automaton F (Figure 2.1.b) represents a factory, which can produce the products (p_1 or p_2) and then deliver it (d_1 or d_2 in synch with C). The "goal" is to deliver products; hence, we mark states where products have been delivered (c_0 and f_0).*

In Figure 2.1.c we depict the parallel composition $C\|F$. Observe that F cannot produce another product until the previous one has been delivered. Thus, the composition $C\|F$ can reach a deadlock – preventing the goal – if a product of the wrong type is produced (i.e., states $\langle c_1, f_2\rangle$ and $\langle c_2, f_1\rangle$). In Figure 2.1.d we show a supervisor S – depicted as an automaton – that avoids deadlock states and guarantees controllability and non-blockingness.

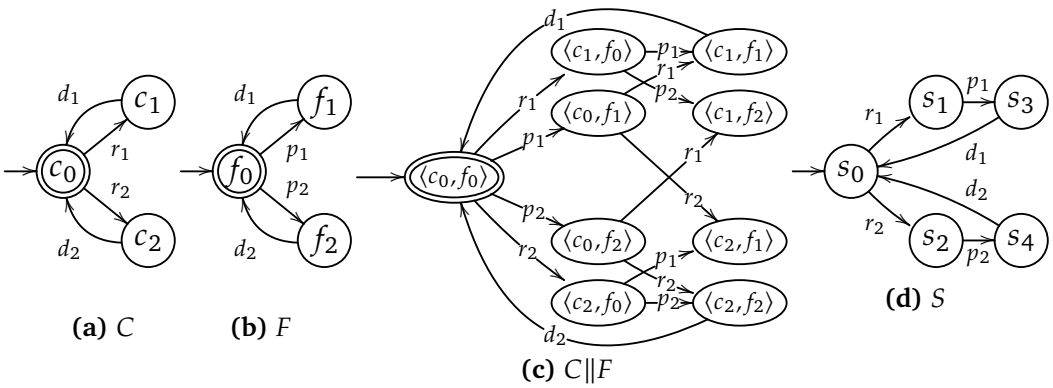


Figure 2.1. Supervisory control problem example.

2.4.2 Reactive Synthesis

Reactive synthesis is concerned with the realization of a reactive module from a specification given in a temporal logic, and assuming an adversarial environment. In this document we take the formalization from [6] and consider the *realizability* of CTL* specifications over a set of Boolean variables V, where V is partitioned in a set of *input* signals In and a disjoint set of *output* signals Out.

Definition 4 (CLT* Formula)**.** A *CLT* formula* is either a *state formula* or a *path formula*, where a state formula is defined inductively using the *path quantifier* $\mathbf{E}$ (possibly; meaning that there exists a path) as follows:

$$\Psi = \mathit{true} \mid v \mid \mathbf{E}\,\Phi \qquad \text{with } v \in V \text{ and } \Phi \text{ a path formula}$$

And a path formula is defined inductively using the standard Boolean connectives plus *termporal operators* $\mathbf{X}$ (next) and $\mathbf{U}$ (until) as follows:

$$\Phi = \Psi \mid \neg\Phi \mid \Phi \vee \Phi \mid \mathbf{X}\,\Phi \mid \Phi\ \mathbf{U}\ \Phi \qquad \text{with } \Psi \text{ a state formula}$$

Additionally, as it is usual, we consider *false*, $\wedge$, $\Rightarrow$, $\mathbf{A}$ (inevitably; meaning for all paths), $\mathbf{G}$ (globally or always) and $\mathbf{F}$ (finally or eventually) as syntactic sugar:

$$\begin{aligned}
\mathit{false} &\Leftrightarrow \neg\mathit{true} \\
\Phi \wedge \Psi &\Leftrightarrow \neg(\neg\Phi \vee \neg\Psi) \\
\Phi \Rightarrow \Psi &\Leftrightarrow \neg\Phi \vee \Psi \\
\mathbf{A}\,\Phi &\Leftrightarrow \neg\,\mathbf{E}\,\neg\Phi \\
\mathbf{G}\,\Phi &\Leftrightarrow \neg\,\mathbf{F}\,\neg\Phi \\
\mathbf{F}\,\Phi &\Leftrightarrow \mathit{true}\ \mathbf{U}\ \Phi
\end{aligned}$$

The semantics of a CTL* formula are defined with respect of a Kripke structure $K = (S, R, L)$, where S is the set of states, $R \subseteq S \times S$ is a total transition relation, and $L : S \mapsto 2^V$ is a function that labels states with the set of variables true in the state. A path in the structure K from a state s_0 is an infinite sequence of states $p = s_0, s_1, \ldots$, such that $(s_i, s_{i+1}) \in R$ for all $i \geq 0$.

We consider the following notation:

- $\Pi(K, s)$ denotes the set of all paths in the structure K from a state s;
- p_i denotes the i-th element of path p;
- $p|^j$ denotes the suffix of path p starting at s_j;
- $K, s \models \Psi$ denotes that Ψ holds at state s if Ψ is a state formula; and
- $K, p \models \Phi$ denotes that Φ holds along path p if Φ is a path formula.

The relation $\models$ is defined inductively as follows:

$$
\begin{array}{lcl}
K, s \models \mathit{true} & & \\
K, s \models v & \Leftrightarrow & v \in L(s) \\
K, s \models \mathbf{E}\,\Phi & \Leftrightarrow & \exists\, p \in \Pi(K, s)\ .\ K, p \models \Phi \\
K, p \models \Psi & \Leftrightarrow & K, p_0 \models \Psi \\
K, p \models \neg\Phi & \Leftrightarrow & \neg(K, p \models \Psi) \\
K, p \models \Phi_1 \vee \Phi_2 & \Leftrightarrow & (K, p \models \Phi_1) \vee (K, p \models \Phi_2) \\
K, p \models \mathbf{X}\,\Phi & \Leftrightarrow & K, p|^1 \models \Phi \\
K, p \models \Phi_1\ \mathbf{U}\ \Phi_2 & \Leftrightarrow & \exists\, j\ .\ (K, p|^j \models \Phi_2) \wedge \\
 & & \forall\, 0 \leq k < j\ .\ (K, p|^k \models \Phi_1)
\end{array}
$$

Realizability for reactive synthesis problems amounts to checking whether there exists an open controller that satisfies the CTL* specification. The realizability problem is best seen as a *two-player game* [6]. The game is played in turns, where player $\mathcal{I}$ starts by giving a subset of the variables in *In* and player $\mathcal{II}$ responds with a subset of the variables in *Out*. The players play according to *strategies*. A strategy for player $\mathcal{I}$ is a mapping $\delta_{\mathcal{I}} : (2^{Out})^* \mapsto 2^{In}$ and a strategy for player $\mathcal{II}$ is a mapping $\delta_{\mathcal{II}} : (2^{In})^* \mapsto 2^{Out}$.

Strategies $\delta_{\mathcal{I}}$ and $\delta_{\mathcal{II}}$ induce the structure $K^{\delta} = (S, R, L)$, where:

- a state $s \in S$ is a finite sequence $s_0, \ldots, s_k$ (denoted $s_{[0,k]}$) such that $s_i \in 2^V$ for all $0 \leq i \leq k$;

- the transition relation R is such that $(s_{[0,k]}, s_{[0,k+1]}) \in R$ if and only if $s_{k+1} = \delta_{\mathcal{I}}(s_0 \backslash In, \ldots, s_k \backslash In) \cup \delta_{\mathcal{II}}(s_0 \backslash Out, \ldots, s_k \backslash Out)$; and
- the labeling function is $L(s_{[0,k]}) = s_k$.

Definition 5 (CTL* Realizability)**.** A CTL* formula Φ over a set of variables V partitioned in intput and output signals (i.e., $V = In \dot{\cup} Out$) is *realizable* if there exists a strategy $\delta_{\mathcal{II}} : (2^{In})^* \mapsto 2^{Out}$ for the system (i.e., player $\mathcal{II}$) such that for every possible strategy $\delta_{\mathcal{I}} : (2^{Out})^* \mapsto 2^{In}$ for the environment (i.e., player $\mathcal{I}$), the structure K^{δ} induced by $\delta_{\mathcal{I}}$ and $\delta_{\mathcal{II}}$ satisfies Φ.

Realizability for general CTL* formulas is known to be a difficult problem (i.e., 2EXPTIME-complete [6]). However, it is possible to bypass this complexity by restricting to the CTL subset of formulas, in which temporal operators must be immediately preceded by a path quantifier. It can be shown that CTL has effectively less expressive power than CTL*. For example, the formula $\mathbf{A}(\mathbf{F}\,\mathbf{G}\,v)$, which expresses that along every path there is a state from which v will hold forever, cannot be expressed in CTL. Still, CTL is sufficiently expressive to capture supervisory control goals, and hence we can leverage on EXPTIME synthesis procedures instead [19].

Nevertheless, differences between supervisory control and reactive synthesis present challenges to be addressed in order to be able to reduce a problem of the former into the latter, specifically:

1. The execution model in reactive synthesis is synchronous, that is environment and system execute in lockstep (changes in the state of signals are synchronized by a clock). Whereas in supervisory control the execution model is asynchronous, that is there is no assumption on the relative execution speeds of the components, which synchronize only through message passing.
2. In reactive synthesis multiple input signals can change simultaneously at a given clock tick (as long as the update is consistent with the problem assumptions). In contrast, in supervisory control only one of the enabled uncontrollable events may ensue from a given state.

2.4.3 Automated Planning

Automated planning is a model-based approach to the synthesis of plans involving the execution of actions (also called operators) that bring about a given goal in a domain. The dynamics of the domain is specified with a factored representation describing the preconditions and effects of actions, together with the goal to be achieved. We focus here on the Fully Observable Non-Deterministic (FOND) variant [58], which extends classical planning with non-deterministic effects, needed to model reactivity.

Definition 6 (FOND Planning)**.** A *FOND planning problem* is a tuple $\mathcal{P} = (F, I, O, G)$, where:

- F stands for the problem *fluents* (i.e., Boolean propositions whose value changes due to action execution);
- $I \subseteq F$ is a set (conjunction) of fluents encoding the *initial situation*;
- O is a set of *operators*; and
- $G \subseteq F$ is a set (conjunction) of fluents defining the *goal* to be achieved.

An *operator* is a pair $o = (Pre(o), Eff(o))$, where $Pre(o)$ is a Boolean formula over F describing the *preconditions* of the operator, and $Eff(o)$ is a conjunction of non-deterministic effects. A *non-deterministic effect* has the form $e_0 \mid \ldots \mid e_n$, where each e_i is a conjunction of deterministic conditional effects (when n=1 the effect is said to be deterministic). A *conditional effect* has the form $C \Rightarrow E$, where C is a Boolean formula over F and E is a *literal* of F (i.e., f or $\neg f$).[2]

A *situation* (also called *state*) is a subset of F (or conjunction of fluents) representing those fluents that are true (in the state). Whenever an operator o with a non-deterministic effect $e_1|\cdots|e_n$ is executed, one of the e_i effects will ensue non-deterministically, that is, without the control of the executor. In turn, a conditional effect $C \Rightarrow E$ states that when condition C holds – in the situation in which the action is being executed – the set of literals E ought to hold in the successor state (and everything else remains static). With this un-

[2] This formalization of actions corresponds to the UC Normal Form [59] with no nested conditional effects.

derstanding, it is possible to define a function $Succ(o, s)$ denoting the *possible successor states* when operator o is executed in a state s [59].

A solution to a FOND planning task is a *policy* $\pi : 2^F \mapsto 2^O$ that maps situation s to a set of appropriate actions $\pi(s)$ such that the goal is eventually reached [60]. A policy is *closed* if it returns an action for every non-goal state potentially reached by following the policy. That is, a policy π is a solution for a planning problem $\mathcal{P} = (F, I, O, G)$ if and only if π is closed, and a sequence of situations $s_0, s_1, \ldots$, such that $s_0 = I$ and for all $0 \leq i \leq k$ it holds that $s_{i+1} \in Succ(o_i, s_i)$ where $o_i \in \pi(s_i)$, is guaranteed to reach a situation $s_k \subseteq G$ (i.e., a goal situation).

Interestingly, different assumptions and solution concepts are studied in planning. A *strong plan* is a closed policy guaranteeing the goal in a bounded finite number of steps in spite of non-determinism, whereas a *strong cyclic plan* is a closed policy guaranteeing the goal in a potentially infinite number of steps [61]. That is, a strong cyclic plan is appropriate in the presence of fairness assumptions, while strong plans guarantee reaching the goal without the need for such assumptions. For the sake of this thesis, we shall focus on strong cyclic plans since they better reflect the non-blocking requirement. That is, we want plans in which the goal is always reachable, but not necessarily reached in a finite number of steps.

Essential differences between supervisory control and planning present a number of challenges to be addressed in order to be able to reduce one problem into the other, namely:

1. Uncontrollable events cannot be modeled explicitly in planning, and need to be modeled through non-determinism.
2. The aim in planning is to reach some *final* state; hence, in our context such a goal needs to identify a situation where a lasso-loop is closed [24] (i.e., finitely guaranteeing that words can always be extended to reach marked states).

2.5 Resumen del capítulo 2

Un acercamiento frecuente para relacionar distintos campos que abordan problemas similares es utilizando traducciones entre sus respectivas formalizaciones. Por lo tanto, en la literatura se han propuesto traducciones entre control supervisor, síntesis reactiva y planificación. No obstante, ninguno de los acercamientos existentes intenta reducir el problema de control supervisor *aprovechando los aspectos composiciones de sus especificiones*.

Las descripciones modulares son centrales al control supervisor, y por lo tanto en esta tesis contribuimos a relacionar los distintos campos proveyendo una traducción que tenga en cuenta la composición. Es decir, nuestra traducción evita la construcción del modelo semántico potencialmente exponencial capturando explícitamente los efectos de la composición en paralelo y la sincronización de componentes en tiempo polinomial. Además discutimos sobre acercamientos con características similares a nuestro trabajo.

Cabe destacar que múltiples herramientas de software han sido desarrolladas para resolver el problema de síntesis. Las técnicas tradicionales pueden ser clasificadas de varias maneras. Por ejemplo, el tipo de entrada que aceptan, si realizan un análisis composicional o monolítico, si mantienen una representa simbólica o explícita, o si realizan una exploración exhaustiva o "sobre la marcha". No obstante, todas las herramientas comparten la característica de que aceptan descripciones compactas.

Las herramientas de control supervisor como MTSA [34] y SUPREMICA [35] toman como entrada autómatas con estados marcados. Las descripciones basadas en autómatas son modulares ya que los distintos componentes intervinientes pueden ser descritos independientemente y luego combinados mediante composición paralela. En el caso de MTSA los autómatas son descritos en un lenguaje de alto nivel de abstracción llamado Finite State Process (FSP). En esta tesis describimos los problemas de control usando scripts FSP, ya que nos permiten crear especificaciones declarativas y escalables.

Las herramientas de síntesis de sistemas reactivos como MBP [37], PARTY [38] y SLUGS [39] toman como entrada fórmulas en lógicas temporales. El estudio de la síntesis reactiva a llevado a la caracterización de fórmulas que

permiten aplicar eficientes soluciones algorítmicas ad-hoc, tales como CTL [40] y GR(1) [20]. Frecuentemente los problemas encontrados en la práctica no requieren la expresividad completa de las lógicas temporales, y por lo tanto los algoritmos eficientes para casos específicos contribuyen significativamente a lidiar con el problema de explosión del espacio de estados.

Las herramientas de planificación como GPT [41], MYND [42] y PRP [43] toman una entrada escrita en el Planning Domain Definition Language (PDDL) [44], que representa un estándar de-facto. En PDDL los problemas son expresados en una forma factorizada usando acciones con precondiciones y efectos, junto con un estado inicial y un objetivo a alcanzar.

En su formalización más general, el control supervisor considera una planta potencialmente no-determinista. Ha sido mostrado que los problemas no-deterministas pueden ser reducidos al caso determinístico. Por lo que restringir el análisis al caso determinístico es una simplificación comúnmente encontrada en la literatura.

En resumen, la síntesis de estrategias operacionales es un problema con muchas facetas. En esta tesis consideramos el problema de síntesis bajo distintos contextos, y comparamos los acercamientos tomados en las tres disciplinas resaltando sus diferencias. Además, proponemos una extensión a nuestro método DCS que evita la explosión inducida por el paso de determinización, lo que permite lograr una mayor escalabilidad.

3

Case Studies

3.1 Motivational examples and benchmark

In order to contextualize supervisory control techniques, we need good motivational examples highlighting the discipline's versatility. Furthermore, if we are to evaluate our proposed algorithmic solutions and compare results with other tools, we also need a rich and diverse set of examples. Unfortunately, there is no such collection of examples publicly available.

A noteworthy collection of examples has been gathered for the 9th International Workshop on Discrete Event Systems (WODES'08), where a benchmark of three classical problems has been proposed. In addition to this benchmark, case studies can be found bundled with supervisory control tools (described in the tools' proprietary specification languages). However, these collections are not amenable for comparing different approaches, since specifications cannot be easily used as input for different tools. Furthermore, the sizes of problem instances are not easily parametrized in the specification languages used, which hinders scalability evaluations. For these reasons, in this chapter we present supervisory control case studies, which serve as motivation and as a benchmark to compare the performance and scalability of different tools.

We build upon the benchmark presented in WODES'08 and enrich it with three additional problems extracted from the literature. We focus on case studies that are scalable, since we would like to test the applicability of the different techniques with respect to the state explosion problem. In particular, we choose problems that scale up in two different directions, namely the number of intervening components and the number of states per component. We do this in order to test how the tools behave, not only as the number of automata grows, but also as the complexity of each intervening automata increases. Additionally, we intentionally include cases that are not realizable for some combination of parameters, since this is reported to be a worst case scenario for some techniques.

In the following sections we give a brief description for each problem in the benchmark. All case studies are written in the FSP language, which allows to scale the number of intervening components (with parameter n) and the number of states per component (with parameter k). FSP is a processes calculus (in the spirit of CSP [36]), which includes standard constructs such as action prefix (->), external choice (|), hiding (\) and parallel composition (||). A detailed explanation of FSP can be found in the Appendix A.

The MTSA tool allows to edit FSP scripts and compile them into standard formats for encoding automata. As part of this thesis, we extend MTSA to include the translations proposed in Chapter 4 to reactive synthesis and planning, which allows us to use the benchmark with tools from these fields. In MTSA, controllable events and marked states are not declared within the processes but in the declaration of the synthesis problem. In order to mark states we use properties definitions related to the occurrence of events. Therefore, together with the processes definitions we present the set of controllable events and the properties marking states.

By default, automata have all their states marked except for the special ERROR state, which is used to model errors and hence is always unmarked. We consider an auxiliary automaton K with two states, one marked and the other unmarked. Transitions through "marking" events lead to the marked states, while any other event produces a step leading to the unmarked state. Therefore, the compositional supervisory control problem requires to keep the plant within its marked language further constrained by K.

Example 2. *Recall the example from Figure 2.1 where two automata C and F model a manufacturing plant that, upon request, produces one of two products. In Listing 3.1 we show how the example can be described in FSP.*

Automaton C, modeling a customer, is described by a process that makes a choice between two uncontrollable options, the request of one of two products r_1 or r_2. Depending on the choice, C must then wait for the delivery of the corresponding product, that is, d_1 or d_2 respectively. Only after receiving the delivery can then C make a new request.

Automaton F, modeling a factory, is described by a process that has the (controllable) choice between producing one of two products p_1 or p_2. Then, depending on that choice, F must deliver the corresponding product (i.e., d_1 or d_2) before being able to produce again.

The plant we wish to control is the composition $C \| F$, yet we avoid its computation. The goal of the system is to deliver products, and hence we would like to mark states where products have been delivered. To do this we indicate a set of "marking" events, that is, we reach a mark state immediately after these events. In this case we mark states after an occurrence of d_1 or d_2 (by default automata have all their states marked except for the special `ERROR` state).

```
C = (r1 -> d1 -> C |
     r2 -> d2 -> C ).

F = (p1 -> d1 -> F |
     p2 -> d2 -> F ).

problem Factory = {
  plant = {C,F}
  controllable = {p1,p2,d1,d2}
  marking = {d1,d2}
}
```

Listing 3.1. Factory problem example in FSP.

3.2 Transfer Line

The Transfer Line (TL), first introduced by Wonham [62], is one of the most traditional examples in supervisory control. This case study is representative of the automated manufacturing domain, which is a common domain of interest for supervisory control.

The TL consists of n machines $M(1), \ldots, M(n)$ connected by n buffers $B(1), \ldots, B(n)$ with finite capacity k and ending in an additional machine called Test Unit (TU). A machine $M(i)$ takes work pieces from the buffer $B(i-1)$ (with the exception of machine $M(1)$ that takes the work pieces from the outside world). And after an undetermined amount of time, the working machine $M(i)$ outputs a processed work piece through buffer $B(i)$. Finally, when a work piece reaches the TU it can be accepted and taken out of the system or it can be rejected and placed back in buffer $B(1)$ for reprocessing.

The goal of the system is to process elements avoiding buffer overflows and underflows. The system controls when to input an element from buffer $B(i-1)$ into machine $M(i)$ (assuming infinite supply from the outside world for $i = 0$); whereas the output from machine $M(i)$ into buffer $B(i)$ is uncontrollable. A catastrophic error ensues if a machine tries to take a work piece from an empty buffer or if it tries to place a processed work piece in a full buffer.

In Listing 3.2 we present the FSP script modeling the TL case study.

Note that we do not require the controller to achieve accepted pieces as acceptance and rejection are not decided by the controller. Instead we mark states where either acceptance or rejection has occurred.

Parameter n represents the number of interconnected machines and buffers. Since machines and buffers are modeled with automata, the number of intervening components scales with n. Parameter k represents the maximum number of work pieces a machine can process simultaneously, plus the maximum number of elements buffers can hold. Since these capacities are modeled by states within the respective automata, the number of states per automata grows with k.

```
M(Id=0) = Working[0],
  Working[wl:0..K] =
    (when (wl < K) get[Id]    -> Working[wl+1] |
     when (wl > 0) put[Id+1] -> Working[wl-1] ).

TU = Idle,
  Idle    = (get[N] -> Testing),
  Testing = (return[1] -> reject -> TU |
             accept -> TU)
            +{return[0..Machines]}.

B(Id=0) = Operative[0],
  Operative[c:0..K] = (
    when (c >  0) get[Id]    -> Operative[c-1] |
    when (c == 0) get[Id]    -> ERROR          |
    when (c <  K) put[Id]    -> Operative[c+1] |
    when (c == K) put[Id]    -> ERROR          |
    when (c <  K) return[Id] -> Operative[c+1] |
    when (c == K) return[Id] -> ERROR          ).

problem TL = {
  plant = {TU, Machine[0..N-1], Buffer[1..N]}
  controllable = {get[0..N]}
  marking = {accept,reject}
}
```

Listing 3.2. TL case study in FSP.

3.3 Dinning Philosophers

The Dinning Philosophers (DP) case study is a classic concurrency problem [63] where n philosophers sit around a table sharing one fork with each adjacent philosopher. The goal of the system is to control the access to the forks allowing each philosopher to alternate between eating and thinking (i.e., avoiding deadlock and starvation). Additionally, after grabbing a fork a philosopher needs to take k intermediate etiquette steps before eating.

We model philosophers and forks as a individual processes (i.e., automata). The only controllable events allow a philosopher to take one fork located adjacent to his position. Marked states are those where philosophers have eaten. In order to determine whether all philosophers have eaten we add auxiliary `Monitor` processes with the `eat.all` event. Note that the problem has no solution when $n = 1$ since there is only one fork on the table.

In Listing 3.3 we present the FSP script modeling the DP case study.

```
 def LeftP(p)  = p
 def RightP(p) = (p+1) % N

 def LeftF(f)  = f == 0 ? N-1 : f-1
 def RightF(f) = f

 Philosopher(Id=0) = Idle,
   Idle = (think[Id] -> Hungry),
   Hungry = (take[Id][LeftP(Id)] -> Etiquete[K]),
   Etiquete[0] = Ready,
   Etiquete[s:1..Steps] = (step[Pid] -> Etiquete[s-1]),
   Ready = (
     take[Id][RightP(Id)] ->
     eat[Id] ->
     release[Id][LeftP(Id)] ->
     release[Id][RightP(Id)] -> Idle).

Fork(Fid=0) = OnTable,
  OnTable = (
    take[LeftF(Fid)][Fid]     -> OnHand  |
    take[RightF(Fid)][Fid]    -> OnHand  ),
  OnHand  = (
    release[LeftF(Fid)][Fid]  -> OnTable |
    release[RightF(Fid)][Fid] -> OnTable ).

 Monitor(Id=0) = (eat[Id] -> Done),
   Done = (eat[Id] -> Done | eat.all -> Monitor).

 problem DP = {
   plant = {Philosopher[0..N-1], Fork[0..N-1], Monitor[0..N-1]}
   controllable = {take[0..N-1][0..N-1]}
   marking = {eat.all}
 }
```

Listing 3.3. DP case study in FSP.

Parameter n represents the number of philosophers and forks. Since philosophers and forks are modeled with automata, the number of intervening components scales with n. Parameter k represents the number etiquette steps a philosopher has to perform before eating. Since etiquette steps are modeled by states within each philosopher automaton, the number of states per automata grows with k.

3.4 Cat and Mouse

The Cat and Mouse (CM) case study [4] is a simple game where n cats and n mice are placed in opposite ends of a corridor divided in $2k + 1$ areas. They take turns to move to one adjacent area at a time, with mice moving first. The goal of the system is to control the movement of mice in order to avoid sharing an area with a cat (the movements of cats are uncontrollable). In the center of the corridor there is a mouse hole, allowing the mice to share the area with the cats safely. Thus, a winning strategy for the mice requires all of them to be always closer to the mouse hole than any cat, in order to be able to evade them.

We model cats and mice with independent processes, where the only controllable events are the movement of mice. Additionally, we use an auxiliary automaton which keeps track of states where a cat and a mouse have been in the same area. Marked states are those where no mouse has shared an area with a cat, except for the central area which is safe for the mice.

In Listing 3.4 we present the FSP script modeling the CM case study.

Parameter n represents the number of cats and mice. Since cats and mice are modeled with automata the number of intervening components scales with n. Parameter k is related to the maximum number of areas in the corridor. Since areas are modeled by states within the cats and mice automata, the number of states per automata grows with k.

```
const Areas = 2*K + 1
const Init = 0
const Last = Areas - 1
const Center = Areas / 2
range Area = Init..Last

Mouse(Mid=0) = Wait[Areas-1],
  Next[a:Area] = (cat.turn -> Wait[a]),
  Wait[a:Area] = (
    mouse.turn -> Act[a] |
    cat[0..N-1].move[b:Area] ->
      if (a==b && a!=Center) then ERROR
      else Wait[a] ),
  Act[a:Area] = (
    mouse[Mid].move[a] -> Next[a] |
    when (a+1 <= Last) mouse[Mid].move[a+1] -> Next[a+1] |
    when (a-1 >= 0   ) mouse[Mid].move[a-1] -> Next[a-1] ).

Cat(Cid=0) = Next[0],
  Next[a:Area] = (mouse.turn -> Wait[a]),
  Wait[a:Area] = (
    cat.turn   -> Act[a] |
    mouse[0..N-1].move[b:Area] ->
      if (a==b && a!=Center) then ERROR
      else Wait[a] ),
  Act[a:Area] = (
    cat[Cid].move[a] -> Next[a] |
    when (a+1 <= Last) cat[Cid].move[a+1] -> Next[a+1] |
    when (a-1 >= 0)    cat[Cid].move[a-1] -> Next[a-1] ).

problem CM = {
  plant = {Mouse[0..N-1], Cat[..N-1]}
  controllable = {mouse[0..N-1].move[Area]}
  marking = {cat.turn}
}
```

Listing 3.4. CM case study in FSP.

3.5 Bidding Workflow

The Bidding Workflow (BW) case study [64] models a company that evaluates projects in order to decide whether to bid for them or not. In order to do this, a document describing a project needs to be accepted by n teams with different specializations. The goal is to synthesize a workflow that attempts to reach consensus, that is, approving the document when all teams accept it, and discarding it when all teams reject it. The document can be reassigned to a team that has rejected it up to k times for revision, but it must not be

```
Team(Id=0) = Pending,
  Pending = (
    {approve,refuse} -> ERROR  |
    assign[Id] -> Assigned[1] ),
  Assigned[s:1..K] = (
    reject[Id][s] -> Rejected[s] |
    accept[Id]    -> Accepted    ),
  Rejected[s:1..K] = (
    refuse  -> Pending |
    approve -> ERROR   |
    assign[Id] ->
      if (s < K) then Assigned[s+1]
      else ERROR ),
  Accepted = (
    {approve,refuse} -> Pending |
    assign[Id] -> ERROR ).

Document = Count[0],
  Count[c:0..Teams-1] = (
    reject[0..N-1][K] -> Rejected   |
    accept[0..N-1]    -> Count[c+1] |
    approve -> ERROR |
    refuse  -> if (c==0) then Document
               else ERROR ),
  Count[N] = (
    {accept[0..N-1],reject[0..N-1][K]} -> Count[N] |
    approve -> Document |
    refuse  -> ERROR    ),
  Rejected = (
    {accept[0..N-1],reject[0..N-1][K]} -> Rejected |
    approve -> ERROR    |
    refuse  -> Document ).

problem BW = {
  plant = {Document, Team[0..N-1]}
  controllable = {assign[Team], approve, refuse}
  marking = {approve, refuse}
}
```

Listing 3.5. BW case study in FSP.

reassigned to a team that has already accepted it. When a team rejects the document k times it can be discarded even with no consensus. This case study represents a problem from the Business Process Management (BPM) domain.

We model the document and the teams with independent processes, where the only controllable events allow assigning the document to a given team for evaluation. We mark states where the document has been approved or dis-

carded. Note that the document process may enter a deadlock if it is approved or discarded without reaching a consensus (or before obtaining k rejections from the same team). Therefore, illegal closures prevent reaching a marked state from that point forward, and hence violating the non-blocking objective.

In Listing 3.5 we present the FSP script modeling the BW case study.

Parameter n represents the number of teams. Since teams are modeled with automata, the number of intervening components scales with n. Parameter k represents the maximum number of rejections a team can emit about a document before discarding it without consensus. Since counting the number of rejection is done with states within the automata, the number of states per automata grows with k.

3.6 Air-Traffic Management

The Air-Traffic Management (AT) case study [65] represents an airport control tower that receives up to n simultaneous requests from planes trying to land. The tower needs to signal planes whether it is safe to approach the landing ramp or at which of k air-spaces they must perform holding maneuvers. If two planes enter the same ramp or air-space a crash may occur. The goal is to control the air-traffic guaranteeing that all the planes can land safely. Notably, the problem only admits solutions when $k > n$, since if there are more landing requests than air-spaces where to hold planes, there is no certain way to avoid a crash.

We model airplanes with processes, where controllable events signal permissions to descend to a certain height or to approach the landing ramp. We use auxiliary automata `HeightMonitor` and `RampMonitor`; to keep track of the usage of the ramp and the air-spaces, observing a crash if it occurs. Additionally, auxiliary automaton `ResponseMonitor` encodes the requirement that the control tower must respond to an airplane request immediately. We mark states where airplanes have landed safely.

In Listing 3.6 we present the FSP script modeling the AT case study.

Note that in this example we make the simplifying assumption that planes have infinite fuel. This allows us to accept supervisors not guaranteeing land-

```
range Plane  = 0..N-1
range Height = 0..K-1

Airplane(Id=0) = (
    requestLand[Id] -> descend[Id][h:Height] -> Holding[h] ),
  Holding[h:Height] =
    if (h>0) then (descend[Id][h-1] -> Holding[h-1])
             else (approach[Id] -> land[Id] -> Airplane).

HeightMonitor(H=0) = Empty,
  Empty = (
    descend[p:Plane][H] -> Occupied[p] |
    when (H>0) descend[Plane][H-1] -> Empty ),
  Occupied[p:Plane] = (
    foreach [o:Plane]
      when (p != o) descend[o][H] -> air.crash[H] -> ERROR |
      when (p != o) descend[o][H-1] -> Occupied[p]  |
      when (p == o && H>0) descend[p][H-1] -> Empty |
      when (p == o && H==0) land[Plane]    -> Empty ).

RampMonitor(R=0) = Empty,
  Empty = (
    approach[Plane] -> Occupied ),
  Occupied = (
    approach[Plane] -> land.crash -> ERROR |
    land[Plane]     -> Empty ).

ResponseMonitor = (
  requestLand[p:Plane] -> descend[p][Height] ->
    ResponseMonitor |
  {descend[Plane][Height], approach[Plane]} ->
    ResponseMonitor ).

problem AT = {
  plant = {
    Plane[Plane], HeightMonitor[Height],
    ResponseMonitor, RampMonitor }
  controllable = {descend[Plane][Height], approeach[Plane]}
  marking = {land[Plane]}
}
```

Listing 3.6. AT case study in FSP.

ings in a bounded amount of steps. Still, the stronger requirement of effectively guaranteeing landings within a given number of steps could be modeled by considering additional automata in the composition.

Parameter n represents the number of airplanes. Since airplanes are modeled with automata the number of intervening components scales with n. Parame-

ter k represents the number air spaces available for holding maneuvers. Since keeping track of occupied air spaces is done with states within the automata, the number of states per automata grows with k.

3.7 Travel Agency

The Travel Agency (TA) case study [66] represents an on-line service that sells vacation packages by relying on existing third-party web-services for different amenities (e.g. car rental, flight purchase, hotel booking, etc.) The goal of the system is to orchestrate the web-services in order to provide a complete vacation package when possible, while avoiding to pay for incomplete packages. This example is representative of the web-service composition problem.

The agency receives requests for vacation packages and interacts with n different web-services for the provision of individual amenities. The protocols for the services may vary (uncontrollably). One variant is the selection of up to k attributes (e.g. flight destination, dates, and class). Another variant in service protocols is that some services may require a reservation step which guarantees a purchase order for a short period, while others do not, and hence the purchase may fail (e.g. on low availability, reservation may be disabled in order to maximize concurrency between clients, and as a result a race condition between two purchase orders may make one fail).

In Listing 3.7 we present the FSP script modeling the main components of the TA case study, in Listings 3.8 we present the FSP script of auxiliary processes.

We use auxiliary automata `ServiceMonitor` and `AgencyMonitor`, which keep track of the procedure followed to assemble a package and signal an error when the goal is not fulfilled. For example, the monitors prevent the agency from informing a client that the package cannot be assembled before querying all services. Additionally, the monitors also prevent the agency from returning an incomplete package (i.e., missing at least one amenity).

Parameter n represents the number of services. Since services are modeled with automata, the number of intervening components scales with n. Parameter k is related to the number of selection steps. Since selection steps are

modeled by states within the services automata, the number of states per automata grows with k.

Observe that modeling such a problem is not an easy task. Moreover, a solution for this problem is not trivial since it should delay payments until all services are assured, (i.e., it has a reservation for each of them). In the case that all but one service are reserved, it should first attempt to secure the uncertain order. If the purchase fails, it should then cancel the reservations and report a failure to the client. In the case that more than two services require direct purchase, it should not attempt to purchase any and return a failure.

```
range Amenity  = 0..N-1
range Step     = 0..K-1

Agency = (agency.request -> Processing),
  Processing = (
    {agency.fail,agency.succ} -> Agency |
    query[Amenity] -> Processing ).

Service(Id=0) = (
  {agency.succ,agency.fail} -> Service |
  query[Id] -> (
    unavailable[Id] ->
      query.fail[Id] -> Service   |
    available[Id] ->
      steps[Id][s:Step] ->
      query.succ[Id] -> Selection[s] )),
  Selection[s:Step] =
    if (s>0) then (select[Id] -> Selection[s-1])
    else Booking,
  Booking  = (
    committed[Id]   -> Reserve |
    uncommitted[Id] -> Direct  ),
  Reserve  = (
    reserve[Id] -> (
      {agency.succ,agency.fail} -> Service |
      cancel[Id]   -> Service |
      purchase[Id] ->
        purchase.succ[Id] -> Service )),
  Direct  = (
    order[Id] -> (
      {agency.succ,agency.fail} -> Service |
      cancel[Id]   -> Service |
      purchase[Id] -> (
        purchase.succ[Id] -> Service |
        purchase.fail[Id] -> Service ) )).

problem TA = {
  plant = {
    Agency, AgencyMonitor, Service[Amenity],
    ServiceMonitor[Amenity] }
  controllable = {
    agency.succ,agency.fail,query[Amenity],reserve[Amenity],
    order[Amenity],purchase[Amenity],cancel[Amenity] }
  marking = {agency.succ, agency.fail}
}
```

Listing 3.7. TA case study in FSP.

```
ServiceMonitor(Id=0) = (
    query[Id]    -> InQuery         |
    agency.succ -> ERROR            |
    agency.fail -> ServiceMonitor ),
  InQuery = (
    query.succ[Id] -> QuerySucces    |
    query.fail[Id] -> QueryFail      |
    agency.succ    -> ERROR          |
    agency.fail    -> ServiceMonitor ),
  QuerySucces = (
    purchase.succ[Id] -> Success        |
    purchase.fail[Id] -> QueryFail      |
    cancel[Id]        -> QueryFail      |
    agency.succ       -> ERROR          |
    agency.fail       -> ServiceMonitor ),
  QueryFail = (
    query[Id]    -> ERROR           |
    agency.succ -> ERROR            |
    agency.fail -> ServiceMonitor ),
  Success = (
    query[Id]    -> ERROR           |
    agency.succ -> ServiceMonitor   |
    agency.fail -> ERROR           ).

AgencyMonitor = Disallow[0],
  Disallow[n:0..1] = (
    agency.fail -> ERROR          |
    agency.succ -> AgencyMonitor  |
    query.fail[Amenity] -> Allow  |
    order[Amenity] ->
      if (n==0) then Disallow[1]
      else Allow  |
    when (n==1) purchase.fail[Amenity] -> Allow ),
  Allow = (
    {agency.fail,agency.succ} -> AgencyMonitor  |
    {query.fail[Amenity],order[Amenity],
     purchase.fail[Amenity]} -> Allow ).
```

Listing 3.8. TA auxiliar processes.

3.8 Resumen del capítulo 3

De forma tal de contextualizar las técnicas de control supervisor necesitamos buenos ejemplos motivacionales que resalten la versatilidad de la disciplina. Además, si pretendemos evaluar nuestra solución algorítmica y comparar resultados con otra herramientas, también necesitamos de un conjunto de ejemplos rico y diverso. Desafortunadamente, no hay tal colección de ejemplos públicamente disponible.

Una destacable recolección de ejemplos fue reunida para el noveno congreso internacional de sistemas de eventos discretos (WODES'08), donde un benchmark de tres problemas clásicos fue propuesto. Además de estos problemas, distintos casos de estudio han sido incluidos por herramientas de control supervisor. Sin embargo, estas colecciones no resultan aplicables a comparar acercamientos diversos, dado que las especificaciones no pueden ser fácilmente usadas como entrada para las distintas herramientas. Además, los tamaños de las instancias de los problemas no son fácilmente parametrizables en los lenguajes de especificación usados, lo que obstaculiza la evaluaciones de escala. Por estos motivos, en este capítulo presentamos casos de estudio de control supervisor que sirven como motivación y benchmark para comparar el desempeño y escalabilidad de las distintas herramientas.

Enriquecemos el benchmark presentado en WODES'08 con tres problemas adicionales extraídos de la literatura. Nos concentramos en casos de estudio escalables, dado que nos interesa validar la aplicabilidad de las distintas técnicas con respecto al problema de explosión del espacio de estados. En particular, elegimos problemas escalables en dos direcciones, la primera representa el número de componentes intervinientes y la segunda el número de estados por componente. Hacemos esto en pos de verificar cómo se comportan las distintas herramientas, no sólo a medida que crece el número de autómatas, sino que también a medida que aumenta la complejidad de cada autómata. Adicionalmente, incluimos casos que no son realizables para alguna combinación de sus parámetros, dado que es un escenario de peor caso para algunas técnicas.

A continuación damos una breve descripción de cada problema. Todos los

casos de estudio están escritos en el lenguaje FSP, que permite aumentar el número de componentes y la cantidad de estados por componente.

- Transfer Line (TL). Uno de los ejemplos más tradicionales del área de control supervisor. Este caso es representativo del dominio de fabricación automatizada. El ejemplo consiste en n máquinas conectadas por n buffers con una capacidad finita k y terminado en una máquina distinguida llamada *TU*, que puede aceptar o rechazar una pieza procesada. El objetivo del sistema es procesar los elementos evitando sobrecargar los buffers.

- Dinning Philosophers (DP). Un problema clásico de concurrencia donde n filósofos se sientan al rededor de una mesa compartiendo un tenedor con cada filósofo adyacente. El objetivo del sistema es controlar el acceso a los tenedores permitiéndole a cada filósofo alternarse entre comer y pensar (evitando deadlocks y starvation). Adicionalmente, luego de tomar un tenedor un filósofo tiene que realizar k pasos intermedios de etiqueta antes de poder comer.

- Cat and Mouse (CM). Es un juego simple donde n gatos y n ratones son ubicados en esquinas opuestas de un corredor divido en $2k + 1$ celdas. Tomando turnos pueden moverse a un área adyacente, con los ratones moviendo primero. El objetivo del sistema es controlar el movimiento de los ratones de forma tal de evitar que comparta una celda con un gato (el movimiento de los gatos es no-controlable). En el centro del corredor hay un agujero de ratón, que permite a los ratos compartir el área con los gatos de forma segura. Por lo que una estrategia ganadora para los ratos requiere que todos ellos siempre se mantengan más cerca del agujero que cualquier gato.

- Bidding Workflow (BW). Este ejemplo modela una compañía que debe evaluar un proyecto para decidir si licitarlo o no. Para ello, un documento describiendo el proyecto necesita ser aceptado por n equipos con distintas especializaciones. El objetivo es sintetizar un flujo de trabajo que persiga lograr un consenso, es decir, que se apruebe el documento cuando todos los equipos lo aceptan y que se descarte cuando todos los equipos lo rechazan. El documento puede ser reasignado a un equipo

que lo haya rechazado hasta k veces para revisión, pero no puede ser reasignado a un equipo que ya lo haya aceptado. Cuando un equipo rechaza el documento k veces, éste puede ser descartado sin consenso.

- Air-Traffic Management (AT). Este caso de estudio representa la torre de control de un aeropuerto que recibe hasta n pedidos simultáneos de aviones queriendo aterrizar. La torre debe indicarle a los aviones cuando es seguro aproximarse a la pista de aterrizaje o en cuál de k espacios aéreos realizar maniobras de holding. Si dos aviones entran en la misma pista o espacio aéreo puede ocurrir una colisión. El objetivo es controlar el tráfico aéreo garantizando que todos los aviones aterricen de forma segura.
- Travel Agency (TA). Este caso de estudio representa una servicio on-line de venta de paquetes turísticos que funciona orquestando servicios web existentes (por ejemplo para alquiler de auto, compra de pasaje aéreo o reservación de hotel). El objetivo del sistema es orquestar los servicios web de forma tal de proveer un paquete vacacional completo cuando sea posible, evitando pagar por paquetes incompletos.

4

Supervisory Control via Reactive Synthesis and Automated Planning

4.1 Overview and Assumptions

In this chapter we show how reactive synthesis and automated planning can be leveraged effectively to solve supervisory control problems for DES. Our hypothesis are: (a) supervisory control problems can be encoded in the reactive synthesis and planning frameworks without incurring in an exponential blowup; and (b) such translations allow leveraging techniques from these other fields to solve compositional supervisory control problems efficiently. Thus, we propose translations of the supervisory control problem into the reactive synthesis and planning frameworks. And, we report on an experimental evaluation comparing the efficacy and efficiency of different tools from the three disciplines. The results show that our translations allow to transparently apply techniques from reactive synthesis and automated planning with an efficiency that rivals that of native supervisory control tools.

In this section we start by clearly stating our assumptions in order to contextualize the proposal within the existing body of knowledge. Specifically, we consider the *standard progress assumption*, which states that the plant will eventually produce one of its available events. In particular, from fully uncontrollable states (i.e., a state from where only uncontrollable events are available) an event must necessarily occur, and hence the plant cannot cause a deadlock in such states.

This assumption brings as a consequence that, in mixed states (i.e., states with controllable and uncontrollable events available), a supervisor may disable all controllable events and wait for the plant to produce an uncontrollable event. In fact, mixed states represent a race condition that, in worst case, can always be won by the plant, and hence if there is a solution to the control problem, it must necessarily have a solution on the occurrence of the uncontrollable events.

Remarkably, diverse assumptions and acceptance conditions, which have an impact on realizability, have been studied in the different fields [67, 68, 24]. Nonetheless, algorithmic approaches under these varied assumptions are usually small adaptations of each other. For instance, dropping the standard progress assumption would only require us to consider fully uncontrollable states as potential deadlocks, and request a supervisor to always enable at least one controllable event from a mixed state. Other adaptations would include different acceptance conditions, such as safety and general liveness goals.

4.2 Supervisory Control via Reactive Synthesis

In this section we develop a translation from a deterministic supervisory control problem (Definition 3) to a reactive synthesis problem (Definition 5) complying with CTL restrictions. Consider $\mathcal{E} = (\{E_0, \ldots, E_n\}, A_C)$ a compositional supervisory control problem, with each $E_i = (S_{E_i}, A_{E_i}, \rightarrow_{E_i}, \bar{e}^i, M_{E_i})$; and $A_C \subseteq A_E$ a partition of the event set into controllable (A_C) and uncontrollable events (A_U). The main features of the translation are:

1. Each transition relation $\rightarrow_{E_i}$ is modeled separately, *avoiding the computation of the parallel composition*.
2. Composition rules are modeled explicitly.
3. An input signal s_i per each automaton E_i is used to keep track of the state in which E_i is at.
4. An input signal u is used – by the environment – to select uncontrollable events to execute.
5. An output signal c is used – by the system – to select events to execute, either controllable or uncontrollable. Remarkably, this does not grant more control to the solution strategy, since it still ought to consider every possible uncontrollable choice.

We encode the supervisory control problem as a CTL formula of the form:

$$\textbf{Initial Condition} \wedge \big(\textbf{Assumptions} \Rightarrow \textbf{Guarantees} \wedge \textbf{Goal}\big)$$

Relying on the following elements.

Variables. The set of variables to be used in the formula are partitioned in input and output signals, such that $\mathit{In} = \mathsf{S} \cup \{\mathsf{u}\}$ and $\mathit{Out} = \{\mathsf{c}\}$, where:

- $\mathsf{S} = \{\mathsf{s}_0, \ldots, \mathsf{s}_n\}$, with each $\mathsf{s}_i \in S_{E_i}$ a variable encoding the state E_i is at.
- $\mathsf{u} \in A_U \cup \{\lambda\}$, a variable encoding the environment choice of an uncontrollable event in A_U plus a special fresh value λ that represents no choice.
- $\mathsf{c} \in A_E$, a variable encoding the system choice of an event in A_E.

Note that for simplicity we consider variables over finite sets, which can be automatically encoded using Boolean variables only.

Notation 2 (Equality)**.** Given an element e from a finite set (of size N), we denote by $\mathsf{s} = e$ the encoding of e as a conjunction of ($\lceil \log_2(N) \rceil$) Boolean variables. That is, characterizing e as a sequence $e_0, \ldots, e_n$ of bits (i.e., Boolean constants), we encode e as the conjunction of n signals $\mathsf{s}^0, \ldots, \mathsf{s}^n$, where each s^i appears negated (i.e., $\neg\mathsf{s}^i$) when e_i is *false*, and positive when e_i is *true*.

Notation 3 (Belongs)**.** We may abuse notation and denote by $\mathsf{s} \in S$ the fact that variable s encodes a state contained in a finite set of states S. More formally: $\mathsf{s} \in S \Leftrightarrow \bigvee_{e \in S} (\mathsf{s} = e)$.

Initial Condition. We initialize variables representing states (s_i) to the initial states ($\bar{e}^i$) of the intervening components:

$$\bigwedge_{i=0}^{n} (\mathsf{s}_i = \bar{e}^i)$$

Goal. We require marked states – of the composed system – to be always reachable (i.e., for all paths there always exists a path where a marked state is eventually reached):

$$\mathbf{AG}\left(\mathbf{EF}\left(\bigwedge_{i=0}^{n} (\mathsf{s}_i \in M_{E_i})\right)\right)$$

Note that marked states in the composition belong to the cartesian product of the marked states of the intervening components (Definition 2). Thus, for a state of the composition to be marked, the states of all individual components have to be marked.

Macros. We use a series of expressions that capture the different aspects of a DES, which expand to simple formulas:

$$ready(\ell, E_i) = \quad \mathsf{s}_i \in \{e \mid e \xrightarrow{\ell}_{E_i} e'\}$$

$$enabled(\ell) = \bigwedge_{\{E_i \in E \mid \ell \in A_{E_i}\}} ready(\ell, E_i)$$

$$step(\ell, E_i) = \begin{cases} \bigwedge_{(e,\ell,e') \in \rightarrow_{E_i}} \big((\mathsf{s}_i = e) \Rightarrow \mathbf{AX}(\mathsf{s}_i = e')\big) & \text{if } \ell \in A_{E_i} \\ \bigwedge_{e \in S_{E_i}} \big((\mathsf{s}_i = e) \Rightarrow \mathbf{AX}(\mathsf{s}_i = e)\big) & \text{otherwise} \end{cases}$$

Observe that (a) macro $ready(\ell, E_i)$ models if event ℓ can be taken from the current state of E_i; (b) macro $enabled(\ell)$ models if event ℓ is ready for every automaton with ℓ in its alphabet (i.e. it synchronizes); and (c) macro $step(\ell, E_i)$ models a step from the current state of E_i via ℓ (or leaves the state unchanged if $\ell \notin A_{E_i}$). Also, note that only *step* uses a temporal operator (i.e., $\mathbf{AX}$), while the other macros use only Boolean connectives.

In the following, we make full use of the expressiveness of these macros to model the supervisory control problem $\mathcal{E}$ in the reactive synthesis framework. In particular, we will ensure that for every event ℓ and automaton E_i, a step through ℓ is taken in E_i (i.e., $step(\ell, E_i)$) only when ℓ is *enabled*.

Assumptions. We encode the transition relation $\rightarrow_E$, such that when an event is selected (either controllable or uncontrollable), the corresponding step is executed (synchronously in each automaton).

(1) $\mathbf{AG}\big((\mathsf{u} = \ell) \Rightarrow step(\ell, E_i)\big)$ for each $E_i \in E$ and $\ell \in A_U$, asserts that if variable u is set to select the uncontrollable event ℓ then variable s_i should be updated as by performing a step in E_i.

(2) $\mathbf{AG}\big((\mathsf{c} = \ell) \wedge (\mathsf{u} = \lambda) \Rightarrow step(\ell, E_i)\big)$ for each $E_i \in E$ and $\ell \in A_E$, asserts that if variable c is set to select event ℓ and u is set to λ, then variable s_i should be updated as by performing a step in E_i. Note that c can be set to an enabled uncontrollable event, which represents disabling all controllable events (forcing the environment to act). This does not grant more control to the system since race conditions always favor the environment, and hence a solution strategy still needs to take into account all the environment's options.

Furthermore, we encode the assumption that the environment will only select valid (enabled) uncontrollable events.

(3) $\mathbf{AG}\big((\mathsf{u} = \ell) \Rightarrow ready(\ell, E_i)\big)$ for each $E_i \in E$ and $\ell \in A_U \cap A_{E_i}$, asserts that if variable u is set to select an uncontrollable event ℓ, then ℓ must be ready in the current state of E_i (note that the conjunction of these formulas require ℓ to be *enabled* in order to be eligible).

Guarantees. We encode the requirement that the controller should only select valid controllable events.

(4) $\mathbf{AG}\big((\mathsf{c} = \ell) \Rightarrow ready(\ell, E_i)\big)$ for each $E_i \in E$ and $\ell \in A_{E_i}$, asserts that if variable c is set to select an event ℓ, then ℓ must be ready in the current state of E_i (again the conjunction of these formulas require ℓ to be *enabled* in order to be eligible).

Observe that in worst case (i.e., considering transitions that connect every state through every event) the complexity of the translation is $O\big((\sum_{i=0}^{n}|A_{E_i}|)(\sum_{i=0}^{n}|S_{E_i}|^2)\big)$, since:

- in order to generate transitions, assumptions and guarantees, each event needs to be considered twice (once to determine the validity of selecting the event, and once to force the update of the state as by performing a step); and
- one variable s_i captures the state E_i is at, and variables (u and c) encode the selection of the event to execute.

This scheme of assumptions and guarantees is inspired in the GR(1) [20] restrictions for LTL formulas, for which efficient synthesis procedures have been proposed. Notice that if the environment purposely violates an assumption, the formula is trivially satisfied (since assumptions are the antecedent of an implication). Also, note that reaching a deadlock state makes the guarantee of selecting an enabled event unsatisfiable. This is correct since reaching a deadlock prevents fulfilling the non-blocking requirement.

Interestingly, the translation also respects other restrictions imposed by GR(1), including limited nesting of temporal operators, and only updating variables of *In* in assumptions while updating variables of *Out* in guarantees (i.e., applying the **X** operator). In spite of the points in common, the non-blocking requirement cannot be expressed in GR(1) since it requires the use of the existential path quantifier **E**, not present in LTL. Remarkably, with LTL we can express a stronger goal requiring to effectively reach marked states (i.e, $\mathbf{G}(\mathbf{F}(\mathit{marked}))$), which is also a relevant type of requirement studied in supervisory control theory [69]. Thus, the approach of targeting an LTL solver with this translation is correct but not complete, that is, a stronger solution obtained by an LTL solver would also be a solution for the supervisory control problem. However, not finding a solution would not imply that a non-blocking solution does not actually exists.

Example 3. *We now depict a fragment of our translation for the compositional supervisory control problem* $\mathcal{E} = (\{C, F\}, A_C)$*, where* C *and* F *are the automata depicted in Figure 2.1.a and Figure 2.1.b respectively; plus*

$A_C = \{d_1, d_2, p_1, p_2\}$ is the set of controllable events (i.e., $A_U = \{r_1, r_2\}$ is the set of uncontrollable events). We use the variables s_C and s_F to encode the states automata C and F are at, and variables c and u to encode the selection of controllable and uncontrollable events to execute (if any). For presentation purposes, we restrict attention to formulas predicating over events d_1, p_1 and r_1, formulas considering events d_2, p_2 and r_2 are analogous.

Initial Condition: $(\mathsf{s}_C = c_0) \wedge (\mathsf{s}_F = f_0)$

Goal: $\mathbf{A}\,\mathbf{G}\,\big(\mathbf{E}\,\mathbf{F}(\mathsf{s}_C \in M_C \wedge \mathsf{s}_F \in M_F)\big)$

Assumptions:

- $\mathbf{A}\,\mathbf{G}\,\big((\mathsf{u}{=}r_1) \Rightarrow \mathit{step}(r_1, C)\big)$, *where* $\mathit{step}(r_1, C) = \big((\mathsf{s}_C{=}c_0) \Rightarrow \mathbf{A}\,\mathbf{X}(\mathsf{s}_C{=}c_1)\big)$.
- $\mathbf{A}\,\mathbf{G}\,\big((\mathsf{u}{=}r_1) \Rightarrow \mathit{step}(r_1, F)\big)$, *where* $\mathit{step}(r_1, F)$ *keeps the value of* s_F, *since* $r_1 \notin A_F$.
- $\mathbf{A}\,\mathbf{G}\,\big((\mathsf{c}{=}r_1) \wedge (\mathsf{u}{=}\lambda) \Rightarrow \mathit{step}(r_1, C)\big)$.
- $\mathbf{A}\,\mathbf{G}\,\big((\mathsf{c}{=}r_1) \wedge (\mathsf{u}{=}\lambda) \Rightarrow \mathit{step}(r_1, F)\big)$.
- $\mathbf{A}\,\mathbf{G}\,\big((\mathsf{c}{=}d_1) \wedge (\mathsf{u}{=}\lambda) \Rightarrow \mathit{step}(d_1, C)\big)$, *where* $\mathit{step}(d_1, C) = \big((\mathsf{s}_C{=}c_1) \Rightarrow \mathbf{A}\,\mathbf{X}(\mathsf{s}_C{=}c_0)\big)$.
- $\mathbf{A}\,\mathbf{G}\,\big((\mathsf{c}{=}d_1) \wedge (\mathsf{u}{=}\lambda) \Rightarrow \mathit{step}(d_1, F)\big)$, *where* $\mathit{step}(d_1, F) = \big((\mathsf{s}_F{=}f_1) \Rightarrow \mathbf{A}\,\mathbf{X}(\mathsf{s}_F{=}f_0)\big)$.
- $\mathbf{A}\,\mathbf{G}\,\big((\mathsf{c}{=}p_1) \wedge (\mathsf{u}{=}\lambda) \Rightarrow \mathit{step}(p_1, C)\big)$, *where* $\mathit{step}(p_1, C)$ *keeps the current value of* s_C, *since* $p_1 \notin A_C$.
- $\mathbf{A}\,\mathbf{G}\,\big((\mathsf{c}{=}p_1) \wedge (\mathsf{u}{=}\lambda) \Rightarrow \mathit{step}(p_1, F)\big)$, *where* $\mathit{step}(p_1, F) = \big((\mathsf{s}_F{=}f_0) \Rightarrow \mathbf{A}\,\mathbf{X}(\mathsf{s}_F{=}f_1)\big)$.
- $\mathbf{A}\,\mathbf{G}\,\big((\mathsf{u}{=}r_1) \Rightarrow \mathit{ready}(r_1, C)\big)$, *where* $\mathit{ready}(r_1, C) = (\mathsf{s}_C{=}c_0)$.

Guarantees:

- $\mathbf{A}\,\mathbf{G}\,\big((\mathsf{c}{=}d_1) \Rightarrow \mathit{ready}(d_1, C)\big)$, *where* $\mathit{ready}(d_1, C) = (\mathsf{s}_C{=}c_1)$.
- $\mathbf{A}\,\mathbf{G}\,\big((\mathsf{c}{=}d_1) \Rightarrow \mathit{ready}(d_1, F)\big)$, *where* $\mathit{ready}(d_1, F) = (\mathsf{s}_F{=}f_1)$.
- $\mathbf{A}\,\mathbf{G}\,\big((\mathsf{c}{=}p_1) \Rightarrow \mathit{ready}(p_1, F)\big)$, *where* $\mathit{ready}(p_1, F) = (\mathsf{s}_F{=}f_0)$.
- $\mathbf{A}\,\mathbf{G}\,\big((\mathsf{c}{=}r_1) \Rightarrow \mathit{ready}(r_1, C)\big)$.

Interestingly, a strategy for this specification is a function similar to the one obtained as a solution for the original supervisory control problem (depicted as an automaton in Figure 2.1.d). That is, the solution is consistent with the solution for the original problem formulation. Moreover, the solution is

"strong" in the sense that marked states can be reached in a bounded number of steps, and hence the supervisor can be found by LTL synthesis as well as by CTL synthesis.

The following result enunciates that the encoding is correct, in the sense that it captures the desired control problem.

Theorem 1. Let $\mathcal{E}$ be a compositional supervisory control problem and let $\varphi_{\mathcal{E}}$ the CTL formula obtained by following the above proposed translation from $\mathcal{E}$. Then, there exists a supervisor σ for $\mathcal{E}$ if and only if there exists a strategy δ for $\varphi_{\mathcal{E}}$. In addition, σ can be constructed from δ and vice versa.

Proof. We prove this by induction on the number of automata. We start by considering the monolithic supervisory control problem $\mathcal{M} = (E_0, A_C)$, that is for a single automaton E_0 (or equivalently a singleton set). In this case the above translation yields a CTL formula $\varphi_{\mathcal{M}}$. Let us assume that there exists a supervisor for $\mathcal{M}$, since $\mathcal{M}$ is a deterministic problem then – as proved in [70] – there is director $\sigma_{\mathcal{M}}$ for $\mathcal{M}$ (i.e., a supervisor for $\mathcal{M}$ that enables at most one controllable event for each state). Let $w = \ell_0, \ldots, \ell_k$ be a word in $\mathcal{L}^{\sigma_{\mathcal{M}}}(E_0)$, then there is a sequence of steps over w, restricted by $\sigma_{\mathcal{M}}$ and starting at the initial state $\bar{e}$ of E_0 (where $\bar{e} = e_0^0$):

$$e_0^0 \xrightarrow{\ell_0}_{E_0} \ldots \xrightarrow{\ell_k}_{E_0} e_{k+1}^0$$

Starting from the initial state e_0^0 we can make the following observations for each step $e_i^0 \xrightarrow{\ell_i}_{E_0} e_{i+1}^0$ in the sequence:

- Initially, variable s_0 encodes state e_i^0 (true for state e_0^0 in the initial condition).
- If $\ell_i \in A_U$ then by rules (3) and (4) variables u and c can be set to select ℓ_i since it is *ready* at e_i^0.
- If $\ell_i \in A_C$ then by rule (4) variable c can be set to encode ℓ_i since it is *ready* at e_i^0 and by rule (2) it must be the case that $\mathsf{u} = \lambda$ (otherwise a race condition would allow the environment to act instead).

- Finally, variable s_0 is updated to encode state e^0_{i+1}, by rule (1) if $\mathsf{u} = \ell_i$ or by rule (2) if $\mathsf{c} = \ell_i$ and $\mathsf{u} = \lambda$.

Therefore, a state is reachable in E_0 restricted by $\sigma_{\mathcal{M}}$ if and only if it is also reachable in a path of $\varphi_{\mathcal{M}}$. Then, since $\sigma_{\mathcal{M}}$ guarantees that a marked state can be reached from e^0_k (i.e., non-blocking), the corresponding path satisfies the goal described in $\varphi_{\mathcal{M}}$. Ergo, a strategy $\delta_{\mathcal{M}}$ obtained by extracting the sequence of observed events w from its input, and returning the encoding of $\sigma_{\mathcal{M}}(w)$, is a winning strategy for $\varphi_{\mathcal{M}}$.

We can extract the sequence of observed events by projecting the event encoded by u if it is not λ, or the event encoded by c otherwise. Whereas, encoding $\sigma_{\mathcal{M}}(w)$ can be done by setting $\mathsf{c} = \sigma_{\mathcal{M}}(w)$ when non-empty, or by setting some enabled uncontrollable event otherwise (since none of these events can prevent the goal).

The inverse direction is similar. That is, for any path in the Kripke structure induced by $\varphi_{\mathcal{M}}$ following strategy $\delta_{\mathcal{M}}$, there is a corresponding run in E_0. By rules (3) and (4) only enabled events can be selected for execution. By rule (1) transitions through uncontrollable events occur independently of the value of c, while by rule (2) transitions through controllable events occur only when $\mathsf{u} = \lambda$ (i.e., not a race condition). In other words, the strategy cannot disable uncontrollable events, complying with the controllability requirement. Then, along a path satisfying $\varphi_{\mathcal{M}}$ it is always the case that a marked state is reachable, also meeting the non-blocking requirement. Therefore, given a strategy $\delta_{\mathcal{M}}$ for $\varphi_{\mathcal{M}}$, we can build a supervisor $\sigma_{\mathcal{M}}$ by extracting the sequence of uncontrollable events w from $\sigma_{\mathcal{M}}$'s input and returning the event encoded by $\delta_{\mathcal{M}}(w)$ if controllable, or an empty set if uncontrollable.

For the inductive case we assume the property holds for the compositional supervisory control problem $(\{E_0, \ldots, E_{n-1}\}, A_C)$. Hence, the compositional problem is equivalent to the monolithic problem over the automaton $E = E_0 \| \ldots \| E_{n-1}$. We use this to check that the property holds for the original set plus E_n, much like in the base case.

Let $\mathcal{E} = (\{E, E_n\}, A_C)$ be a compositional supervisory control problem and $\varphi_{\mathcal{E}}$ its translation as a reactive synthesis problem. Let $\sigma_{\mathcal{E}}$ be a director for $\mathcal{E}$ and let $w = \ell_0, \ldots, \ell_k$ be a word in $\mathcal{L}^{\sigma_{\mathcal{E}}}(E \| E_n)$. Then there is a sequence of

steps over w in $E \| E_n$ starting in the initial state $\langle \bar{e}, \bar{e}^n \rangle$ (where $\bar{e} = e_0$ and $\bar{e}^n = e_0^n$):

$$\langle e_0, e_0^n \rangle \xrightarrow{\ell_0}_{E\|E_n} \dots \xrightarrow{\ell_k}_{E\|E_n} \langle e_{k+1}, e_{k+1}^n \rangle$$

Starting from the initial state $\langle e_0, e_0^n \rangle$ we can make the following observations for each step $\langle e_i, e_i^n \rangle \xrightarrow{\ell_i}_{E\|E_n} \langle e_{i+1}, e_{i+1}^n \rangle$ in the sequence:

- Initially variable s encodes state e_i and variable s_n encodes state e_i^n (true for $\langle e_0, e_0^n \rangle$ in the initial condition).
- If $\ell_i \in A_U$ then by rules (3) and (4) variables u and c can be set to select ℓ_i since it is *ready* in both e_i and e_i^n (i.e., *enabled*).
- If $\ell_i \in A_C$ then by rule (4) variable c can be set to encode ℓ_i since it is *ready* in both e_i and e_i^n (i.e., *enabled*); and by rule (2) it must be the case that $\mathsf{u} = \lambda$ (i.e., not a race condition).
- Finally, s is updated (as by performing a *step*) to e_{i+1} and s_n is updated to e_{i+1}^n; this is done for E and E_n by rule (1) if $\mathsf{u} = \ell_i$, or by rule (2) if $\mathsf{c} = \ell_i$ and $\mathsf{u} = \lambda$.

As before, a state is reachable in $E \| E_n$ if and only if it is reachable in a path of $\varphi_{\mathcal{E}}$, and we can build a strategy $\delta_{\mathcal{E}}$ from a director $\sigma_{\mathcal{E}}$.

The inverse direction is analogous to the base case. Still, we highlight that the conjunction of rules (3) and (5) for E and E_n restricts selection of events to those effectively enabled. And, the conjunction of rules (1) and (2) for E and E_n effectively model the synchronous execution of events. That is, $\varphi_{\mathcal{E}}$ effectively captures the semantics of $\mathcal{E}$, taking the rules of parallel composition into account. Therefore, once again we can build a supervisor $\sigma_{\mathcal{E}}$ from a strategy $\delta_{\mathcal{E}}$. □

4.3 Supervisory Control via Automated Planning

In this section we develop a translation from a supervisory control problem (Definition 3) to a planning problem (Definition 6). The translation follows the line of the translation to reactive synthesis, but capturing uncontrollable events through non-determinism. The main features of the translation are:

1. Each transition relation $\rightarrow_{E_i}$ is modeled separately, *avoiding the computation of the parallel composition*.
2. Composition rules are modeled explicitly.
3. Each controllable event ℓ_c is mapped to an operator ℓ_c (abusing notation) available to the planner.
4. Each uncontrollable event ℓ_u is mapped to an operator ℓ_u (abusing notation) that the planner *is forced to take* after its selection at a distinguished "non-deterministic choice phase."
5. For simplicity, we do not require the non-deterministic choice of uncontrollable events to choose a valid (enabled) event. Thus, when no valid event is selected we default the choice to the planner. This does not grant more control to the planner, as it still ought to consider (and resolve) every possible non-deterministic choice.
6. We request a *strong cyclic* plan reaching a marked state, and then use the approach presented in [24] to extend the goal with recurrence.

Consider the compositional supervisory control problem $\mathcal{E}$=($\{E_0, \ldots, E_n\}, A_C$), with E_i=($S_{E_i}, A_{E_i}, \rightarrow_{E_i}, \bar{e}^i, M_{E_i}$) for $0 \leq i \leq n$; and $A_C \subseteq A_E$ a partition of the event set into controllable (A_C) and uncontrollable events (A_U). We encode $\mathcal{E}$ as a planning problem $\mathcal{P}_{\mathcal{E}} = (F, I, O, G)$ as follows:

Fluents. The set of fluents is defined as $F = \mathsf{S} \cup \{\mathit{inprogress}(\ell_u) \mid \ell_u \in A_U\} \cup \{\mathit{picked}, \mathit{event}\}$ where:

- $\mathsf{S} = \{\mathsf{s}_0, \ldots, \mathsf{s}_n\}$, where each $\mathsf{s}_i \in S_{E_i}$ is a fluent encoding the state E_i is at (analogous to variables s_i used in the translation to reactive synthesis);
- *event*, a Boolean fluent encoding that the last operator executed represented an event and not an auxiliary operator;
- $\mathit{inprogress}(\ell_u)$, a Boolean fluent encoding that uncontrollable event ℓ_u has been selected for execution at the last non-deterministic choice phase; and

- *picked*, a Boolean fluent encoding that a non-deterministic choice phase has occurred since the last event execution.

For simplicity, we consider fluents over finite sets, which can be automatically encoded using Boolean fluents only.

Initial Situation. The initial situation I captures the initial state $\bar{e}^i$ of each component E_i:

$$I = \bigwedge_{i=0}^{n} (\mathsf{s}_i = \bar{e}^i) \wedge \mathit{event}$$

Goal. The goal specification is to reach a situation representing a marked state (through an event):

$$G = \bigwedge_{i=0}^{n} (\mathsf{s}_i \in M_{E_i}) \wedge \mathit{event}$$

Macros. We use the same macros as in the reactive synthesis translation. However, we restate the macros into the planning framework, that is, relying on conditional effects instead of logical implication and fluent update instead of the *next* temporal operator. Note that contrarily to reactive synthesis, in planning, fluents that are not explicitly updated remain the same, thus we restate the *step* macro taking this into account too:

$$\mathit{step}(\ell, E_i) = \bigwedge_{(e,\ell,e') \in \rightarrow_{E_i}} \big((\mathsf{s}_i = e) \Rightarrow (\mathsf{s}_i = e')\big)$$

Operators. Set $O = A_E \cup \{\mathsf{pick}\}$ is the collection of planning operators where:

- Operator pick non-deterministically picks an uncontrollable event from set $A_U = \{\ell_0, \ldots, \ell_k\}$.

$$Pre(\mathsf{pick}) = \neg picked$$

$$\begin{aligned} Eff(\mathsf{pick}) = picked \wedge \neg event \wedge \\ \big(enabled(\ell_0) \Rightarrow inprogress(\ell_0) \mid \\ \vdots \\ enabled(\ell_k) \Rightarrow inprogress(\ell_k) \mid \\ true \Rightarrow true\big) \end{aligned}$$

Note that pick turns the *picked* fluent on, which prevents repeating its application until the fluent is turned off. Additionally, the non-deterministic choice may select an enabled event, which is then set to be in-progress (i.e., forcing the planner to execute said event). Alternatively, the choice may not set any in-progress fluent by selecting a non-enabled event or the last "dummy" option (allowing the plant to voluntarily loose a race condition even when all the uncontrollable events are enabled).

- Each operator $\ell_c \in A_C$ updates the components states as per the corresponding controllable event ℓ_c. Controllable events can be selected by the planner only when enabled and if no uncontrollable events are in-progress (i.e., effectively resolving race conditions in favor of the environment).

$$Pre(\ell_c) = \bigwedge_{\ell \in A_U} \neg inprogress(\ell) \wedge enabled(\ell_c) \wedge picked$$

$$Eff(\ell_c) = \bigwedge_{\{E_i \in E \mid \ell_c \in A_{E_i}\}} step(\ell_c, E_i) \wedge event \wedge \neg picked$$

- Each operator $\ell_u \in A_U$ updates the components states as per the corresponding uncontrollable event ℓ_u. The planner is forced to select operator ℓ_u if it set to be in-progress (i.e., by pick), or may choose ℓ_u if no event is in-progress (forcing the environment to act as if by disabling all controllable events).

$$Pre(\ell_u) = \Big(inprogress(\ell_u) \vee \bigwedge_{\ell \in A_U} \neg inprogress(\ell)\Big) \wedge enabled(\ell_u) \wedge picked$$

$$Eff(\ell_u) = \bigwedge_{\{E_i \in E \mid \ell_u \in A_{E_i}\}} step(\ell_u, E_i) \wedge event \wedge \neg picked \wedge \bigwedge_{\ell \in A_U} \neg inprogress(\ell)$$

Note that each operator ℓ_u resets all the *inprogress* fluents (not necessary for controllable events, since they are never on in the controllable case). This, together with the reset of the *picked* fluent performed by all operators representing events, re-establishes the initial invariants guaranteeing that in the following: (a) operator pick will be eligible next; (b) an operator ℓ_c – representing an enabled controllable event ℓ_c – will be eligible afterwards if no valid uncontrollable event is selected by pick; and (c) an operator ℓ_u – representing an enabled uncontrollable event ℓ_u – will be eligible afterwards if selected by pick or if no such event is selected.

Finally, applying the approach presented in [24] we extend the reachability planning problem $\mathcal{P}_\mathcal{E}$ into a recurrent planning problem $\mathcal{P}'_\mathcal{E} = (F', I', O', G')$ by requiring to form a lasso-loop after reaching a marked state. This makes necessary to add a new operator loop, which dynamically designates a marked state – of the composition – as the ulterior goal. The modification also calls for including n additional fluents (one per component) to allow the effect of loop to "store" the current state. The extension is as follows:

- $F' = F \cup \{\mathsf{s}'_0, \ldots, \mathsf{s}'_n\}$, the original fluents plus new fluents $\mathsf{s}'_i \in S \cup \{ß\}$ that encode a selected state to be reached for a second time (or a distinguished value ß if a state has yet to be selected).
- $I' = I \wedge \bigwedge_{i=0}^{n} (\mathsf{s}'_i = ß)$, the initial situation extended to set the new s'_i fluents to the distinguished value ß.
- $O' = O \cup \{\mathsf{loop}\}$, the original operators plus the loop operator defined as follows:

$$Pre(\mathsf{loop}) = \bigwedge_{i=0}^{n} (\mathsf{s}_i \in M_{E_i}) \wedge \bigwedge_{i=0}^{n} (\mathsf{s}'_i = \text{ß}) \wedge \mathit{event}$$
$$Eff(\mathsf{loop}) = \bigwedge_{i=0}^{n} \Big(\bigwedge_{e \in M_{E_i}} \big((\mathsf{s}_i = e) \Longrightarrow (\mathsf{s}'_i = e)\big) \Big) \wedge \neg \mathit{event}$$

Observe that operator loop is only eligible in a marked state and if it has not been executed before (i.e., variables s'_i have not been set), and its effect "stores" the state encoded in variables s_i into the auxiliary variables s'_i.

- $G' = \bigwedge_{i=0}^{n} (\mathsf{s}_i = \mathsf{s}'_i) \wedge \mathit{event}$, the goal is to reach a situation representing a marked state previously selected by operator loop (through an event).

A result to the recurrent planning problem will be a strong cyclic policy π with the form $\pi_1, \mathsf{loop}, \pi_2$, where π_1 is a strong cyclic policy that can reach a marked state s from the initial state, loop is the auxiliary operator that dynamically designates s as a goal, and π_2 is a strong cyclic policy that parts from s and can get to back s. Note that since loop turns the *event* fluent off, the planner is forced to take an action to turn the fluent back on. This avoids trivially reaching the target state without taking any action.

It is not difficult to see that in worst case (i.e., considering transitions connecting every state through every event) the complexity of the above translation is $O\big((\sum_{i=0}^{n}|A_{E_i}|)(\sum_{i=0}^{n}|S_{E_i}|^2)\big)$, since:

- in order to generate the operators each event needs to be considered once; and
- fluents are the union of states in S_{E_i}, plus extra fluents added to "store" a reached marked state, labels of A_U for the *inprogress* fluents, and the *picked* fluent.

Example 4. *In Listing 4.1 we show a fragment of our translation in PDDL for the compositional supervisory control problem $\mathcal{E} = (\{C, F\}, A_C)$, where C and F are the automata depicted in Figure 2.1.a and Figure 2.1.b respectively; and $A_C = \{d_1, d_2, p_1, p_2\}$ is the set of controllable events, while $A_U = \{r_1, r_2\}$ is the set of uncontrollable events. Mind that in PDDL predicates and logic operators are applied in prefix notation following a Lisp tradition.*

```
(:init (and (sC=c0) (sF=f0) (event) (sC'=ß) (sF'=ß)))

(:goal (and (sC=sC') (sF=sF') (event)))

(:action pick
  :precondition (not picked)
  :effect (and (picked) (not event)
               (oneof (when (enabled r1) (inprogress r1))
                      (when (enabled r2) (inprogress r2))
                      (when (true) (true)) )))
(:action loop
  :precondition (and (sC = 0) (sF = 0)
                     (sC' = ß) (sF' = ß)
                     (event) )
  :effect (and (when (sC = 0) (sC' = 0))
               (when (sF = 0) (sF' = 0))
               (not event) ))
(:action d1
  :precondition (and (not (inprogress r1))
                     (not (inprogress r2))
                     (enabled d1) (picked) )
  :effect (and (when (sC = c1) (sC = c0))
               (when (sF = f1) (sF = f0))
               (event) (not (picked)) ))
(:action p1
  :precondition (and (not (inprogress r1))
                     (not (inprogress r2))
                     (enabled p1) (picked) )
  :effect (and (when (sF = f0) (sF = f1))
               (event) (not (picked)) ))
(:action r1
  :precondition (and (or (inprogress r1)
                         (and (not (inprogress r1))
                              (not (inprogress r2)) )
                     (enabled r1) (picked) )
  :effect (and (when (sC = c0) (sC = c1))
               (event) (not (picked))
               (not (inprogress r1))
               (not (inprogress r2)) ))
```

Listing 4.1. Planning translation example.

We use the fluents s_C *and* s_F *to encode the states automata* C *and* F *are at, and fluents picked and inprogress*(ℓ) *(for each* $\ell \in A_U$*) to encode the non-deterministic choice of an uncontrollable event. For presentation purposes we focus on events* d_1*,* p_1 *and* r_1*, operators for* d_2*,* p_2 *and* r_2 *are analogous; and while we do not expand macros enabled, we do expand macros step.*

Observe that operator r_1 *only updates the fluent* s_C *(since* $r_1 \notin A_F$*). Moreover,*

operator p_1 only updates the fluent s_F (since $p_1 \notin A_C$). On the contrary, operator d_1 updates s_C and s_F synchronously. Furthermore, r_1 is uncontrollable (i.e., must be in-progress to be eligible, or no other event must be in-progress), while d_1, and p_1 are controllable (i.e., to be eligible no uncontrollable event can be in-progress).

The example also shows auxiliary operators pick, and loop, which encode the non-deterministic choice of an uncontrollable event (relying on the `oneof` keyword for non-deterministic choices) and recurrence requirements.

The expansion of macros enabled boils down to Boolean formulas over fluents s_C and s_F. For instance:

- `enabled d1 ⇔ (and (sC = c1) (sF = f1))`, *that is, for d_1 to occur C has to be at state c_1 and F at state f_1 simultaneously;*
- `enabled p1 ⇔ (sF = f0)`, *that is, p_1 only requires F to be at f_0; and*
- `enabled r1 ⇔ (sC = c0)`, *that is, r_1 only requires C to be at c_0.*

A solution policy for this planning problem is a function similar to the one obtained for the original supervisory control problem (depicted as an automaton in Figure 2.1d. That is, the solution is consistent with the solution for the original problem formulation.

The following result enunciates that the encoding is indeed correct, in that it fully captures the problem of interest.

Theorem 2. Let $\mathcal{E}$ be a compositional supervisory control problem and $\mathcal{P}'_{\mathcal{E}}$ its corresponding planning problem as per the above encoding. Then, there exists a supervisor σ for $\mathcal{E}$ if and only if there exists a strong cyclic policy π for $\mathcal{P}'_{\mathcal{E}}$. In addition, σ can be constructed from π and vice versa.

Proof. This proof follows that of Theorem 1. In the following, we focus on the main differences, namely the validity of the translation for the monolithic case, and the mapping between planning policies and supervisors.

We start by considering the monolithic control problem $\mathcal{M} = (E, A_C)$, that is for a single automaton $E = (S_E, A_E, \rightarrow_E, \bar{e}, M_E)$. In this case the above

translation yields a planning problem $\mathcal{P}$. Let σ be a supervisor for $\mathcal{M}$, which we additionally require to be memoryless, that is, σ has to base its decisions solely on the states of the plant. In other words, for any two words w and w' reaching the same state e it holds that $\sigma(w) = \sigma(w')$, and hence we may abuse notation and consider $\sigma(w) = \sigma(e)$. This is without loss of generality since if there is a solution for a deterministic supervisory control problem there exists a memoryless supervisor [71].

Let $w = \ell_0, \ldots, \ell_k$ be a word of $\mathcal{L}^\sigma(E)$, that is, there is a sequence of steps in E starting at the initial state $\bar{e}$ (where $\bar{e} = e_0$) and restricted by σ:

$$e_0 \xrightarrow{\ell_0}_E \ldots \xrightarrow{\ell_k}_E e_k$$

For every step $e_i \xrightarrow{\ell_i}_E e_{i+1}$ in the sequence we can make the following observations:

- In situation s_i fluent s encodes state e_i, fluent *picked* is off, and fluent *event* is on (true for e_0 in the initial situation $s_0 = I$).
- Since fluent *picked* is off the only eligible action in s_i is pick, which leads to a situation s_i' where *picked* is on, and *inprogress*(ℓ_i) might be on too since ℓ_i is enabled in e_i, or alternatively all *inprogress* fluents might be off.
- If in s_i' fluent *inprogress*(ℓ_i) is on, then ℓ_i is the only eligible action, leading to situation s_{i+1}; in s_{i+1} fluent s encodes e_{i+1}, *picked* is off and *event* is on (i.e., reestablishing the original invariant).
- If in s_i' fluent *inprogress*(ℓ_i) is off (all *inprogress* fluents are off), then ℓ_i can be selected by the planner since it is enabled from e_i, leading again to situation s_{i+1}.

Therefore, a state e_i is reachable in E restricted by σ if and only if e_i is also reachable in $\mathcal{P}$. Supervisor σ guarantees that a marked state e_m can be reached from e_k, consequently in $\mathcal{P}$ a situation s_m representing e_m can be reached from a situation s_k representing e_k. Furthermore, since E is finite we can check whether every word in $\mathcal{L}^\sigma(E)$ reaching marked state e_m can be extended to reach e_m again; let $E_M \subseteq M_E$ be the set of such marked states

(i.e., E_M represents the states from where it is safe to execute loop). Given the fact that S_E is finite, E_M cannot be empty since that would make σ violate the non-blocking property.

A strong cyclic policy π for $\mathcal{P}$ can be derived from σ by mapping states to situations (since σ is memoryless), considering with special care auxiliary operators pick and loop. That is, operator pick must be interleaved between operators representing events; and operator loop must be executed when first visiting a state of E_M (i.e., marked states that can be revisited controllably). When operator pick selects an uncontrollable event ℓ_u (i.e., setting it *inprogress*) then π must return ℓ_u next. Otherwise, for a situation s encoding a state $e \in S_E$, $\pi(s)$ can return $\sigma(e)$. Since σ is non-blocking, every word accepted by $\mathcal{L}^\sigma$ can be extended to reach a marked state, and hence this also holds for π, making it strong cyclic.

The inverse direction is similar. Note that in a strong cyclic policy π for $\mathcal{P}$ operator pick can always choose an enabled uncontrollable event which the planner is forced to take, thus complying with the controllability requirement. This non-deterministic choice may fail to select a valid event, still the policy can enforce the standard progress assumption selecting an enabled uncontrollable event for execution (i.e., forcing the plant to act as by disabling all controllable events). Furthermore, operator loop flags a marked state known to be always reachable, consequently also meeting the non-blocking requirement. Therefore, a memoryless supervisor σ can be derived from π, that is, given a state $e \in S_E$ and a situation s encoding e we can define $\sigma(e)$ by carefully following $\pi(s)$ (i.e., discarding auxiliary operator loop, plus returning the union of results after every possible outcome of operator pick).

The generalization to the compositional case can be proved inductively in the number of automata in the compositional supervisory control problem. The main consideration to take into account is that the *enabled* and *step* macros need to effectively capture the rules of parallel composition in Definition 2. This is straightforward since *enabled* captures the antecedent of rules in Definition 2 while *step* captures their consequent.

□

4.4 Validation

In this section we report on an evaluation of the proposed translations over the benchmark presented in Chapter 3. The aim of the evaluation is twofold: (a) we validate our hypothesis that the translations are indeed efficient; and (b) we test whether our approach, despite its added complexity, successfully allows leveraging on tools from reactive synthesis and planning to solve supervisory control problems. In order to do so, we compare tools native to supervisory control with tools from the other disciplines working under our encoding.

For each of the six problems in the benchmark, we vary parameters affecting the number of components and states per component (n and k respectively) independently between 1 and 6. Hence, the evaluation considers the execution of 36 tests per case study, totaling 216 tests per tool. The number of reachable states in the composed plants varies from the order of 10^1 states, up to the order of 10^{10} states (depending on n, k and the particular topology of each problem).

4.4.1 Threats to validity

On the one hand, we highlight that comparing tools from different disciplines is not an easy task and, in spite of our best effort, it is important to understand that it might not be completely fair. In particular, traditional supervisory control techniques aim at generating maximally permissive supervisors, and despite the fact that the selected tools (i.e., MTSA and SUPREMICA) *do not guarantee maximality* (SUPREMICA offers to do so *optionally*), they may naturally incur in higher computational costs since, in general, they consider multiple controllable options.

On the other hand, we acknowledge that the results of the automatic translations are structured differently and are usually larger than manually written specifications (despite of only growing polynomially). There is a risk that this may hinder the tools working under our encoding. In particular, the selected tools from planning (i.e., MYND and PRP) rely on extracting useful heuristic information from the input knowledge representation. Moreover, subtle mod-

eling alternatives often have great impact on the performance and conformity of the different tools.

The evaluation considers CTL synthesis as supported by MBP, a reactive synthesis tool based on the well-known NuSMV model checker. MBP not only supports CTL specifications but also accepts PDDL inputs in an attempt to apply the symbolic techniques used in reactive synthesis to planning. When given a model in PDDL format, MBP first translates the model to the SMV format and then launches the synthesis procedure. In the presence of reachability goals, MBP uses specialized efficient synthesis techniques. However, this translation from PDDL to SMV may incur in an exponential blowup, especially when faced with complex conditional PDDL constructs. In particular, when given a PDDL generated by our translation, this first step often fails due to this explosion. Thus, we use MBP exclusively as a reactive synthesis tool, efficiently generating the SMV file with our translation and without relying on the specialized optimizations for reachability.

Finally, we consider including in the evaluation the SLUGS tool, an efficient reactive synthesis solver for GR(1) goals. In this particular case our translation is correct but *not complete*, since in GR(1) we can only express a goal stronger than the non-blocking requirement (i.e., marked states are guaranteed to be reached infinitely often). That is, for a given problem if there is a non-blocking solution but no stronger solution, SLUGS will report that no solution exists. In spite of this, we consider the tool because it is representative of a highly related area and might promote further cross-fertilization. In particular, we highlight that the MTSA tool also implements an optimized algorithm for this particular case.

Taking the above into consideration and despite our tuning attempts to make the most out of every tool, this evaluation cannot be used to establish a categorical superiority of one approach over the other. Yet, it suffices to validate our hypothesis on the applicability of compositional translations.

4.4.2 Results

In Table 4.1, we show the time required to perform the translations for each case study. The experiments were run on a desktop computer with an Intel

i7-3770, with 8GB of RAM, and a time-out of 30 minutes. The time required by the translations is negligible compared to the time required to solve the different problems (i.e., input files for all tools not native to supervisory control can be generated in a couple of minutes, while running the benchmark in the selected hardware requires days).

	SMV	SLUGS	PDDL
AT	18.3	18.8	18.7
BW	16.4	16.6	16.7
CM	28.1	28.4	26.8
DP	16.2	16.5	16.6
TA	19.2	19.8	19.6
TL	15.7	15.9	16.1

Table 4.1. Translation time per case study and input format in seconds.

In Figure 4.1 we show results detailed by tool and case study, each plot shows the combination of parameters n (horizontal axis) and k (vertical axis). Black represents a time-out, or out of memory. Interestingly, the memory bound is not an issue for the considered tools except for MTSA that, by building an explicit representation of the composition, is the only tool that runs out of memory.

In Figure 4.2.a we summarize the results reporting on the totals of solved cases (i.e., no time-outs, out of memory or other failures); and in Figure 4.2.b we report on the total execution time in minutes (sum of times of every test up to – and including – time-outs).

From the results we can make the following observations:

- Planning tools (MYND and PRP), having as input the PDDL files obtained through our translation, performed very well. This hints that heuristic search might be a good alternative to solve compositional supervisory control problems.
- Reactive synthesis tools (MBP and SLUGS), also taking as input the the results of our translation, solved a similar number of instances to supervisory control tools but required more time.
- Despite relying on a compositional analysis SUPREMICA performs only

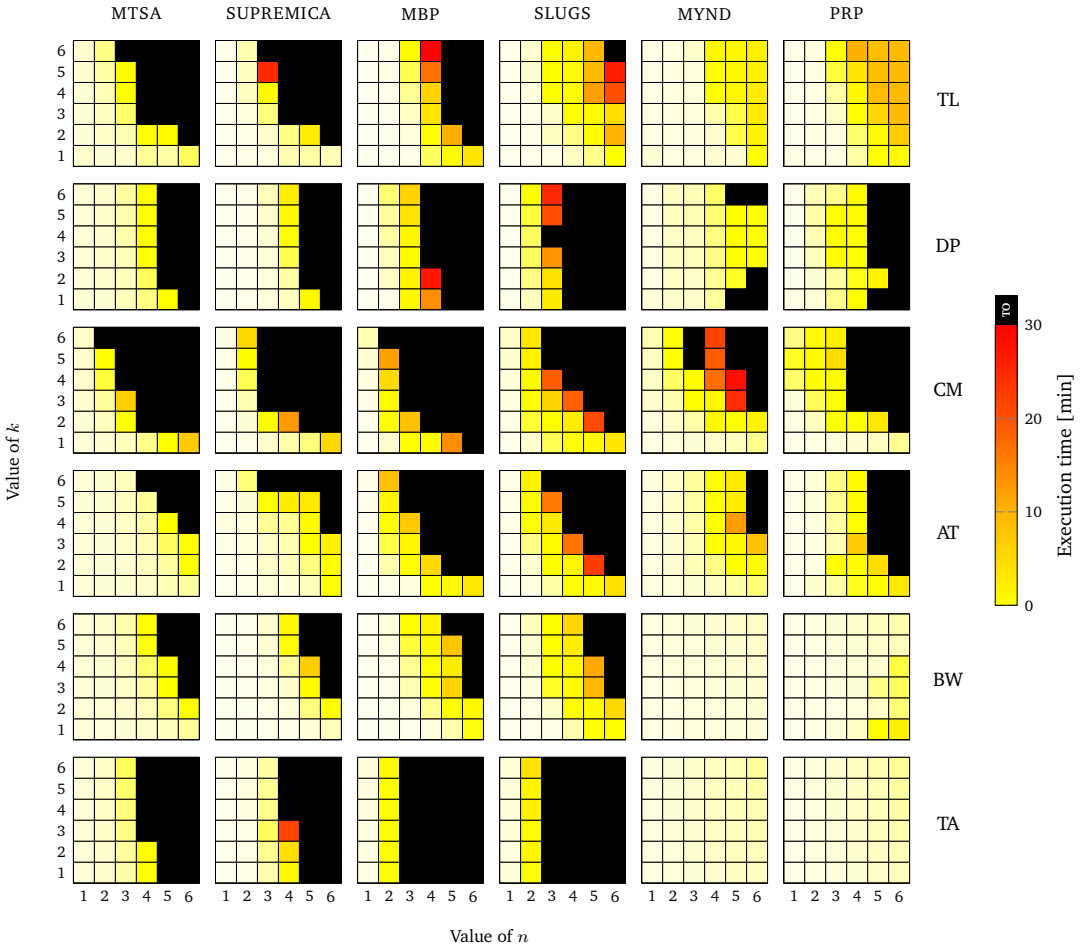
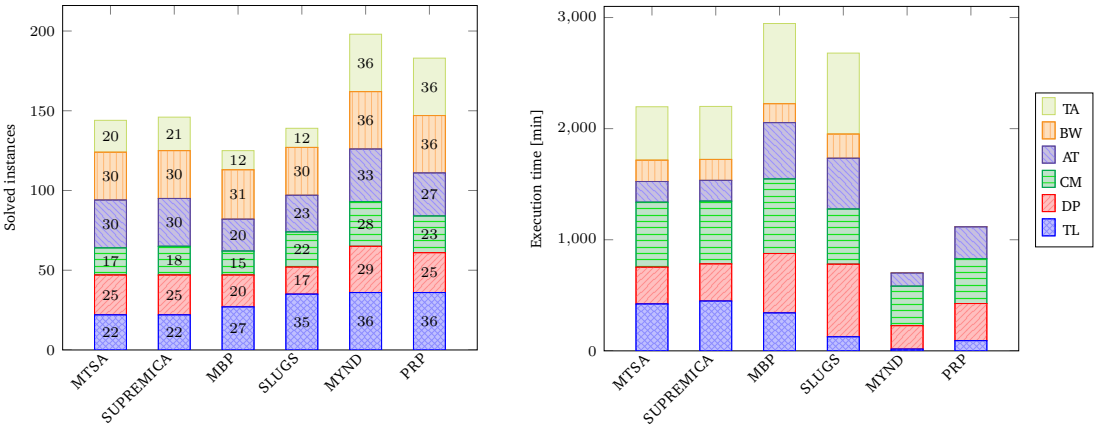


Figure 4.1. Detailed benchmark results.

slightly better than the monolithic approach (implemented by MTSA). Interestingly, in spite of being native to supervisory control, it solves less instances of the benchmark than other tools working under our encoding.

- Around 55% of instances are solved in less than 10 minutes, only 15% are solved between 10 and 30 minutes, while the remaining 30% reaches the time out. This highlights the steep growth in complexity observed as the state space explodes.
- The CM case study proved to be particularly challenging for all tools. Closer inspection revealed that not only the problem space incurs in a blowup, but also the solution grows exponentially.



(a) Number of solved instances.

(b) Total execution time.

Figure 4.2. Summarized benchmark results.

- Scaling up the number of processes n tends to present a bigger challenge than increasing k. This is to be expected since the state space grows as $O(k^n)$.

4.5 Summary and Insights

In this chapter, we present translations from compositional supervisory control problems to reactive synthesis and to planning. The translations are polynomial with respect to the size of a compact input specification, thanks to encoding the compositional aspects of control specifications. The translation-based approach to supervisory control is viable since the computational overhead introduced by the translation is negligible with respect of the cost of solving the synthesis problem.

Applying the translations, we compared tools from the three fields and found that they effectively allow leveraging the advances made in these related areas to solve supervisory control problems. We have found that the best performing tools for the selected benchmark are not tools native to supervisory control, despite the size and structure of the translated specifications. Still, the state explosion problem creates a steep complexity jump that takes tools out of their zone of applicability abruptly.

These results hint that, despite the polynomial growth in input specifications,

heuristic search is a good alternative for solving compositional supervisory control problems. This raises the question of whether the techniques from the different areas can be combined for greater efficacy, keeping both the size of specifications and solutions small. In the next chapter we present a novel algorithmic solution that combines directed on-the-fly exploration with compositional analysis and compare its results with the ones reported above.

4.6 Resumen del capítulo 4

En este capítulo mostramos cómo la síntesis de sistemas reactivos y la planificación automática pueden ser utilizadas efectivamente para resolver problemas de control supervisor de sistemas de eventos discretos. Nuestras hipótesis son: (a) los problemas de control supervisor pueden ser codificados en el marco de síntesis reactiva y planificación sin incurrir en un costo exponencial; y (b) tales traducciones nos permitirían aprovechar las técnicas de estos campos para resolver eficientemente problemas composicionales de control supervisor. Por lo tanto proponemos traducciones del problema de control supervisor en síntesis reactiva y planificación. Presentamos una evaluación experimental comparando la eficacia de diferentes herramientas de las tres disciplinas. Los resultados muestran que nuestras traducción permiten aplicar transparentemente técnicas de síntesis reactiva y planificación automática con una eficiencia competitiva con herramientas nativas a control supervisor.

Empezamos indicando claramente nuestras hipótesis de forma de contextualizar la propuesta dentro del cuerpo de conocimiento existente. Específicamente, consideramos la hipótesis de progreso estándar, que afirma que una planta eventualmente producirá uno de sus eventos disponibles. En particular, a partir de un estado completamente no-controlable (i.e., un estado desde donde sólo se encuentran eventos no-controlables habilitados) un evento debe necesariamente ocurrir, y por consiguiente la planta no puede causar un deadlock en tal estado.

Estas hipótesis tienen como consecuencia que, en estados mixtos (i.e., estados con eventos controlables y no-controlables habilitados), un supervisor puede deshabilitar todos los eventos controlables y esperar a que la planta produzca un evento no-controlable. De hecho, los estados mixtos representan

una condición de carrera que en peor caso siempre puede ser ganada por la planta, y por lo tanto si hay una solución al problema de control necesariamente debe lidiar con la ocurrencia de los eventos no-controlables.

Cabe destacar que diversas hipótesis y condiciones de aceptación, que tienen un impacto sobre la realizabilidad, han sido estudiadas en los distintos campos [67, 68, 24]. No obstante, los acercamientos algorítmicos bajo estas hipótesis variadas son usualmente pequeñas adaptaciones de los tradicionales. Por ejemplo, descartar la hipótesis de progreso estándar sólo requeriría considerar los estados completamente no-controllables como deadlocks potenciales, y requerir un supervisor que siempre habilita al menos un evento controlable en los estados mixtos. Otras adaptaciones incluirían diferentes criterios de aceptación, como safety y liveness.

La traducción de control supervisor a planificación automática tiene las siguientes características principales:

1. Cada relación de transición es modelada independientemente, evitando computar de la composición paralela de los componentes.
2. Las reglas de composición son modeladas explícitamente.
3. Cada evento controlable es representado con un operador disponible al planificador.
4. Cada evento no-controlable es representado con un operador que el planificador está forzado a tomar luego de su selección en una fase distinguida de elección no-determinística.
5. Por simplicidad no requerimos que la elección no-determinística seleccione un evento válido (i.e., habilitado). Por lo que, cuando un evento inválido es seleccionado otorgamos al planificador la elección del siguiente evento a ejecutar. Esto no amplía las capacidades del controlador ya que de todas formas requiere considerar toda posible elección no-controlable.
6. Requerimos un plan cíclicamente fuerte alcanzando un estado marcado y luego usamos el acercamiento presentado en [24] para extender el objetivo con recurrencia.

Ambas traducciones son polinomiales con respecto al tamaño de las especificaciones compactas, gracias a la codificación de los aspectos composicionales de las especificaciones de control. El acercamiento de control supervisor basado en traducciones es viable dado que la sobrecarga computacional introducida por la traducción es negligible con respecto al costo de resolver el problema de síntesis.

Aplicando las traducciones comparamos herramientas de los tres campos y encontramos que efectivamente nos permiten aprovechar los avances hechos en estas áreas relacionadas para resolver problemas de control supervisor. En particular hemos encontrado que las herramientas con mejor desempeño para el benchmark seleccionado no son las herramientas nativas a control supervisor, a pesar del tamaño y la estructura de las especificaciones traducidas. De todos modos el problema de explosión del espacio de estados crea un salto abrupto de complejidad que hace que las herramientas salgan rápidamente de su zona de aplicabilidad.

Los resultados obtenidos dan un indicio que, a pesar del crecimiento polinomial en las especificaciones, la búsqueda heurística es una buena alternativa para resolver el problema de control supervisor. Esto da lugar a la pregunta de si las técnicas de las distintas áreas pueden ser combinadas para lograr una eficacia mayor, manteniendo acotado el tamaño de las soluciones y especificaciones.

5

Directed Controller Synthesis for Deterministic DES

5.1 On-the-fly Directed Exploration

In this chapter we present the Directed Controller Synthesis (DCS) of DES. The DCS method explores the solution space on-the-fly guided by a domain-independent heuristic. In this section, we discuss the main algorithm that constitutes DCS, namely, the on-the-fly exploration. In the following sections, we present heuristics derived from abstractions of the plant, which exploit the componentized way in which complex environments are described. Then, we report on an evaluation comparing DCS to other techniques over the benchmark presented in Chapter 3.

Given a compositional supervisory control problem $\mathcal{E}$ and a heuristic function that estimates the distance from a state of E to a goal (i.e., a marked state in M_E reachable infinitely often), DCS looks for a supervisor by computing

the parallel composition on-the-fly guided by the heuristic. In worst case, the heuristic misguides the exploration and the whole parallel composition is built. However, great savings can be obtained, if the heuristic guides the exploration in such a way that only a reduced portion of the state space is considered.

5.1.1 Exploration Procedure

The on-the-fly exploration procedure is a modification of Best First Search (a classical informed search procedure) adapted to account for uncontrollable events. We perform a forward exploration, keeping a priority queue of *open* states ordered by their estimated distance to a goal, and initialized only with the initial state $\bar{e}$. At each iteration we take the highest "priority" state e from the queue and, by following its first unexplored event, we expand a child state e' adding it to the exploration structure. If we identify e' as a goal or an error we prune the exploration. Otherwise, applying the heuristic to e' we obtain a ranking of its enabled events, and then we insert e' in the *open* queue (considering the "priority" of e' to be the estimated distance of its first unexplored event).

Special considerations need to be taken into account when, after expanding a state e' from a state e, a loop is closed in the exploration structure. This happens when the child state e' has been already expanded in some previous iteration and is an ancestor of e. If such a loop encloses a marked state e_m, it may represent a goal (i.e., if we can controllably extend words ending in e to reach e_m). Additionally, it can also represent a goal if there exists a path out of the loop leading to a marked state. Yet the state space explored up to this point may be insufficient to distinguish between these cases. Thus, we may need to postpone the decision on the status of e until the exploration progresses further.

Therefore, every time a loop is closed by following a transition from a state e to a state e', we need to inspect the exploration structure to decide if the state is a goal, an error, or so far undetermined. In order to do this we check whether e belongs to a Controllably Connected Component (CCC) unfolding from e', that is, a set $\mathbb{C}$ of the states reachable from e' for which: (a) all un-

controllable transitions lead back to $\mathbb{C}$; and (b) for states where there are no enabled uncontrollable events, at least one controllable transition also leads back to $\mathbb{C}$.

Definition 7 (Controllably Connected Component)**.** Given an automaton $E = (S_E, A_E, \rightarrow_E, \bar{e}, M_E)$, and $A_U \subseteq A_E$ a set of uncontrollable events; a set of states $\mathbb{C} \subseteq S_E$ is a *controllably connected component* if and only if:

$$\forall e \in \mathbb{C} \,.\, \{e' \mid e \xrightarrow{\ell_u}_E e' \wedge \ell_u \in A_U\} \subseteq \mathbb{C} \wedge \{e' \mid e \xrightarrow{\ell}_E e'\} \cap \mathbb{C} \neq \emptyset$$

If we discover a CCC $\mathbb{C}$, we then need to consider whether it contains a marked state $e_m \in M_E$ reachable from every state of $\mathbb{C}$, and in such a case, we say that $\mathbb{C}$ is *marked*. If $\mathbb{C}$ is marked, the states in the component can be safely flagged as goals and excluded from further exploration. On the other hand, if $\mathbb{C}$ is not marked, its states remain undetermined since further exploration may reveal a larger CCC including them, and hence they need to be reopened.

The exploration process is repeated until a marked CCC containing the initial state is discovered, or until arriving to the conclusion that such a CCC does not exists. If the *open* queue becomes empty or the initial state is flagged as an error, DCS reports that the problem is unrealizable (i.e., there is no supervisor). Whereas, if the initial state is flagged as a goal (i.e., belongs to a marked CCC) we build a supervisor by following, from the initial state, the transitions connecting states also flagged as goals.

Remarkably, we do not immediately explore all successors upon expanding a new state, we only explore its first unexplored transition instead. Still, in worst case, all transitions are explored and, for this reason, if there is a supervisor the algorithm finds it. This is because, once we determine that a successor state is a goal, an error, or that has been exhaustively explored; we iteratively propagate this "status" information back to its ancestor states. In doing so, we check whether each of these ancestor states require additional exploration or if they can be directly flagged as goals or errors. If a state requires additional exploration, it is placed back in the *open* queue, which does not become empty until all expanded states have been properly examined.

5.1.2 Status Propagation

Exploring a transition from a state e may trigger a status propagation (i.e., *goal*, *error* or *undetermined*). The error status is propagated when a transition leads to a state already flagged as error and when expanding a deadlock state (since we know for certain that this prevents achieving the goal). The goal status is propagated when a transition leads to a state already flagged as a goal and upon discovering a marked CCC. The undetermined status is propagated when closing a loop over a state that does not belong to a marked CCC, but from which there could still be unexplored paths leading to such a goal. We do this in an orderly fashion propagating the status information from e back to its ancestors.

At some point, the propagation may be interrupted and a state reopened. The propagation of an error through a controllable transition is interrupted when reaching an ancestor state with unconfirmed *controllable* events (i.e., events still under consideration). Whereas, error propagation through uncontrollable transitions and into ancestors with unconfirmed *uncontrollable* events does not cause an interruption. This is because the (adversarial) environment can prevent the supervisor from fulfilling its objective by taking the event known to lead to an error from such a state. States from which we have propagated an error are added to a set of identified *Errors*.

The propagation of a goal and undetermined status is interrupted when reaching an ancestor state with unconfirmed *uncontrollable* events. Whereas, ancestors with unconfirmed *controllable* events do not interrupt goal propagation. This is because in order to achieve the objective, it suffices to enable only one such event from these states. If a state is confirmed to belong to a marked CCC, it is added to a set of identified *Goals*, which in turn constitutes a marked CCC. Otherwise, the state is kept undetermined until the exploration progresses further, reopening the state or one of its ancestors. This is similar to decision procedures on AND/OR trees [72], where all children need to hold for AND nodes while just one child suffices for OR nodes.

5.1.3 Heuristic Considerations

While there is an obvious advantage in exploring the most promising controllable event first, there is no such advantage in exploring the best uncontrollable event, since all such events must lead to a goal. Thus, we could avoid the computation of the heuristic for fully uncontrollable states. Instead, we use the reverse ranking and expand the least desirable uncontrollable event first. We do this in an attempt to identify an error as fast as possible, avoiding futile exploration by closing the state early (i.e., similarly to a Minimax [73] approach). For this reason, we require the ranking for mixed states to consist of uncontrollable events in descending order, followed by controllable events in ascending order.

Furthermore, when propagating a goal, we require all uncontrollable events to avoid errors, while holding at least one event (uncontrollable or not) known to lead to a goal. In fully controllable states, we are contented with any controllable event leading to a goal. Thus, in order to broaden the exploration options, we keep states in the *open* queue (i.e. competing with their descendants) as long as they have solely unexplored controllable events. On the contrary, we only reopen a state with remaining unexplored uncontrollable events, on the confirmation that its open descendants do not lead to errors.

Additionally, we highlight that our goal is not simply to reach a marked state, but to always be able to extend words to reach such a state. For this reason we need to find a marked CCC, and hence we require the heuristic to guide the exploration towards a CCC containing at least one marked state.

5.1.4 Algorithm

In Listing 5.1 we present the on-the-fly exploration procedure taking a compositional supervisory control problem and a heuristic function as parameters. In Listing 5.2 we present status propagation procedures, which perform a backwards Breath First Search (BFS) for states that need to be reopened. And finally in Listing 5.3 we present common auxiliary procedures.

For ease of presentation we consider the following:

```
procedure DCS(E=(E, A_C), heuristic):
  initial = ⟨ē^0, ..., ē^n⟩
  openState(initial, heuristic)

  while open ≠ ∅ ∧ initial ∉ Errors ∪ Goals:
    (e, ℓ, e') = expandNext(dequeue(open), recommendations)
    toOpen = ∅
    if e' ∈ Errors:
      toOpen = propagateError(e)
    else if e' ∈ Goals:
      if extendsCCC(e, Goals):
        toOpen = propagateGoal(e)
      else:
        toOpen = propagateUndetermined(e)
    else if isLoop(e,e'):
      ℂ = buildMCCC(e,e')
      if ℂ ≠ ∅:
        Goals = Goals ∪ ℂ
        toOpen = propagateGoal(e)
      else:
        toOpen = propagateUndetermined(e)
    else:
      toOpen = if ℓ ∈ A_C: {e,e'} else: {e'}
    for o ∈ toOpen:
      openState(o, heuristic)

  if initial ∈ Goals:
    return (λw . {ℓ | initial -w->_ES e -ℓ->_ES e' ∧ e' ∈ Goals}) // a supervisor
  else:
    return UNREALIZABLE
```

Listing 5.1. On-the-fly Directed Exploration Procedure.

- We assume the presence of a "global" table called Exploration Structure (*ES*) where the explored state space is recorded. Every time a new state is expanded, it is added to the *ES*, recording the transition leading to it (i.e., source state and event label). We may abuse notation and denote by $e \xrightarrow{\ell}_{ES} e'$ the fact that step $e \xrightarrow{\ell}_{E} e'$ has been recorded in *ES*.
- The heuristic function takes a state e and the *ES*, and returns a ranking of the events enabled from e. Heuristic functions may extract useful information from the *ES*, such as the marked states visited in the path from the initial state to e.
- We keep the results of the application of the heuristic function in a *recommendations* table. Thereby, the evaluation of the heuristic function

```
procedure propagateUndetermined(e):
  visited = ∅
  pending = {e}
  while pending ≠ ∅:
    o = dequeue(pending)
    visited = visited ∪ {o}
    if recommendations[o] ∩ A_U = ∅:
      for p ∈ predecesors(o) such that p ∉ visited ∪ pending:
        enqueue(pending, p)
  return visited

procedure propagateError(e):
  visited = ∅
  pending = {e}
  while pending ≠ ∅:
    o = dequeue(pending)
    visited = visited ∪ {o}
    if forcedToError(o):
      Errors = Errors ∪ {o}
      for p ∈ predecesors(o) such that p ∉ visited ∪ pending:
        enqueue(pending, p)
  return visited \ Errors

procedure propagateGoal(e):
  visited = ∅
  pending = {e}
  while pending ≠ ∅:
    o = dequeue(pending)
    visited = visited ∪ {o}
    if extendsCCC(o, Goals):
      Goals = Goals ∪ {o}
      for p ∈ predecesors(o) such that p ∉ visited ∪ pending:
        enqueue(pending, p)
  return visited \ Goals
```

Listing 5.2. Status propagation procedures.

is done only once per expanded state. When a state is reopened, the next recommendation from the table is consumed.

- The propagation procedures (i.e., `propagateGoal`, `propagateError` and `propagateUnconfirmed`), perform a backwards BFS collecting ancestors states that need to be reopened for further exploration. In the process, states in the path may be flagged as goals or errors if their status can be determined for certain in the *ES*.
- The `enqueue`, `dequeue` and `peek` procedures are implemented over a standard priority queue. We use an optional parameter with the "pri-

```
procedure openState(e, heuristic):
  if e ∉ recommendations:
    recommendations[e] = heuristic(e, ES)
  if recommendations[e] ≠ ∅:
    enqueue(open, e, peek(recommendations[e]))

procedure expandNext(e, recommendations):
  ℓ = dequeue(recommendations[e])
  let e' such that e -ℓ->_E e'
  if isDeadlock(e'):
    Errors = Errors ∪ {e'}
  ES[s][ℓ] = e'
  return (e, ℓ, e')

function isDeadlock(e):
  return ¬∃ℓ . e -ℓ->_E e'

function isLoop(e, e'):
  return ∃w . e' -w->_ES e  // assuming e -ℓ->_ES e'

function buildMCCC(e, e'):
  let ℂ such that
    ℂ = {e_i | (e' -w->_ES e_i -w'->_ES e ∨ e -w->_ES e_i -w'->_ES e') ∧ extendsCCC(e_i, ℂ ∪ Goals)} ∧
    (∃w . e -w->_ES e_m ∧ e_m ∈ M_E ∩ (ℂ ∪ Goals))
  return ℂ // a marked CCC with e' or ∅ if it does not exist

function extendsCCC(e, ℂ):
  return (∃ℓ . e -ℓ->_E e' ∧ e' ∈ ℂ) ∧ (∀ℓ_u ∈ A_U . e -ℓ_u->_E e' ⇒ e' ∈ ℂ)

function forcedToError(e):
  return if (∃ℓ_u ∈ A_U . e -ℓ_u->_E e' ∧ e' ∈ Errors): true
         else: (∀ℓ_c ∈ A_C . e -ℓ_c->_E e' ⇒ e' ∈ Errors)

function predecesors(e'):
  return {e | e -ℓ->_ES e'}
```

Listing 5.3. DCS auxiliary procedures.

ority" of a state, which is given by the estimated distance of its first unexplored recommendation. By default, the priority parameter is 1, which allows us to use it as a standard queue too.

- The `openState` procedure enqueues a state e in the *open* queue using e's first unexplored recommendation as priority. Additionally, the first time e is considered, the procedure populates the *recommendations* table with the result of computing the heuristic function for e.
- The `expandNext` procedure expands a child state e' by consuming the

parent's first unused recommendation. That is, it produces a new state of the composition "on-the-fly" by following the transition indicated by the recommendation. If e' is detected to be a deadlock, it is immediately flagged as an error. Then, e' is added to *ES* and returned.

- The `buildMCCC` function returns a marked CCC (potentially extending *Goals*) with the intermediate states in the paths between two states e and e', known to be in a loop. Since the states in these paths form a Strongly Connected Component (SCC), this can be computed efficiently with a variation of Tarjan's SCC algorithm [74]. That is, we can obtain an SCC $\mathbb{C}$ in linear time with respect to the number of steps, and then remove the states that violate the CCC property from $\mathbb{C}$. We are left with an empty set only when there is no subset of $\mathbb{C}$ constituting a CCC. Thanks to the strong connectivity between states in $\mathbb{C}$ it is easy to see whether $\mathbb{C}$ is marked by checking if there is a path from e to a marked state e_m (i.e., we do not need to check this for every state in $\mathbb{C}$).
- The `extendsCCC` function indicates whether a state e is "controllably closed" with respect to a CCC $\mathbb{C}$. That is, the function returns whether adding e to $\mathbb{C}$ maintains the CCC property.
- The `forcedToError` function indicates whether a state e can be uncontrollably forced into an error. This happens when some uncontrollable transition leads to an error from e or when all controllable transitions from e lead to an error.

One advantage of the proposed algorithm is its simplicity, since it merely computes a fragment of the parallel composition by moving forward from the initial state. When a loop or a deadlock is discovered, the information is propagated backwards through the *ES* flagging or reopening the states in the path. When the propagation reaches the initial state we learn whether a supervisor exists.

One disadvantage of the proposed algorithm is that the heuristic may misguide the search and, in worst case, the algorithm might end up computing the complete parallel composition (i.e., expanding the states after every enabled event). Note that, with the additional cost of computing the heuristic for each expanded state, the complexity of the algorithm is actually worse

than its monolithic counterpart. Thus, DCS could take an exponential amount of time (i.e., EXPTIME).

Example 5. *Recall the example from Figure 2.1 where two automata C and F, represent the interaction between a customer and a factory. Let us assume the existence of an heuristic function h that accurately estimates the distance from a state to the goal. In Figure 5.1, we depict the complete exploration structure built by* DCS *indexing each state with the number of the iteration in which it is expanded.*

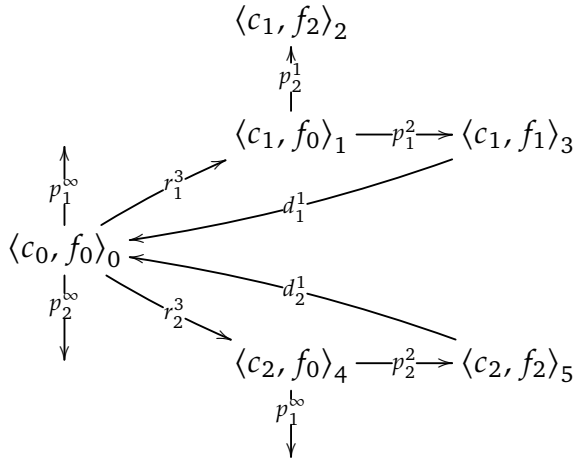


Figure 5.1. Exploration structure built guided by h.

DCS *starts the on-the-fly exploration at state $\langle c_0, f_0\rangle_0$ and evaluates the heuristic function h at the state. The enabled events at $\langle c_0, f_0\rangle_0$ are p_1, p_2, r_1 and r_2; thus h should return a ranking of these events with their estimated distances to the goal. Let us assume that $h(\langle c_0, f_0\rangle) = [r_1^3, r_2^3, p_1^\infty, p_2^\infty]$, where an event's superscript indicates its estimated distance to the goal. Observe that uncontrollable events r_1 and r_2 are prioritized, while controllable events p_1 and p_2 are relegated. Also, note that h estimates an unbounded (∞) distance to the goal for p_1 or p_2, this is accurate because the plant can be uncontrollably led to a deadlock if it produces before receiving a request.*

DCS *continues by following the first event r_1, expanding $\langle c_1, f_0\rangle_1$. The enabled events at this state are p_1 and p_2, which are both controllable. In order to decide which event to analyze first we apply h to $\langle c_1, f_0\rangle_1$. For the sake of*

completeness, let us consider that h misguides the exploration in this case, in particular $h(\langle c_1, f_0\rangle) = [p_2^1, p_1^2]$. That is, h recommends exploring p_2 first, and so DCS *expands $\langle c_1, f_2\rangle_2$ (a deadlock state). The goal cannot be achieved from such a state, and hence* DCS *starts the propagation of an error. The propagation flags $\langle c_1, f_2\rangle_3$ as an error and continues backwards, considering its parent state $\langle c_1, f_0\rangle$. Since p_2 is controllable and $\langle c_1, f_0\rangle_1$ still has unexplored events, the propagation is interrupted.*

DCS *then explores p_1 from $\langle c_1, f_0\rangle_1$, which expands $\langle c_1, f_1\rangle_3$. Let us assume that $h(\langle c_1, f_1\rangle) = [d_1^1]$. As a consequence,* DCS *explores the transition over d_1, reaching the state $\langle c_0, f_0\rangle_0$ and closing a loop. At this point,* DCS *checks whether $\langle c_1, f_1\rangle_3$ belong to a CCC unfolding from $\langle c_0, f_0\rangle_0$ (i.e., considering the states reachable from $\langle c_0, f_0\rangle_0$). However, it is immediate that this is not the case since there is an unexplored uncontrollable event from state $\langle c_0, f_0\rangle_0$ (namely r_2). Therefore, the undetermined status is propagated and $\langle c_0, f_0\rangle_0$ is reopened for further consideration.*

DCS *continues exploring r_2 from $\langle c_0, f_0\rangle_0$, expanding $\langle c_2, f_0\rangle_4$. This time the remaining recommendations from $\langle c_0, f_0\rangle_0$ are all controllable events (i.e., p_1^∞ and p_2^∞), and hence the state remains in the open queue. Let us consider $h(\langle c_2, f_0\rangle) = [p_2^2, p_1^\infty]$. Thus, at this point we have two open states $\langle c_0, f_0\rangle_0$ and $\langle c_2, f_0\rangle_4$, we have recommendation p_1^∞ from the former, while p_2^2 from the latter. That is, pending controllable event p_1 competes with its descendant p_2.* DCS *explores p_2, since $2 < \infty$, which expands $\langle c_2, f_2\rangle_5$.*

Assume now that $h(\langle c_2, f_2\rangle) = [d_2^1]$, that is, DCS *is recommended to explore d_2 (a more promising alternative than p_1^∞ from $\langle c_0, f_0\rangle_0$ and p_1^∞ from $\langle c_2, f_0\rangle$) and in doing so it closes a loop over $\langle c_0, f_0\rangle_0$. This time the state does belong to a CCC unfolding from $\langle c_0, f_0\rangle$ (i.e., the set $\{\langle c_0, f_0\rangle, \langle c_1, f_0\rangle, \langle c_1, f_1\rangle, \langle c_2, f_0\rangle, \langle c_2, f_2\rangle\}$), which in particular is marked, and hence we propagate a goal. When the propagation finishes, the initial state $\langle c_0, f_0\rangle_0$ is detected to be flagged as a goal, and the exploration procedure finishes.*

Building the controller amounts to following transitions leading to states flagged as goals (i.e., all the expanded states except $\langle c_1, f_2\rangle_2$). Observe that all reachable uncontrollable transitions are included in fully uncontrollable

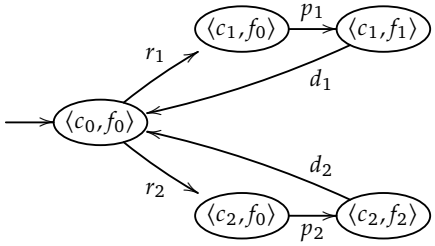


Figure 5.2. Supervisor computed by DCS.

states, and at least one controllable transition is included in fully controllable states. Moreover, in mixed states all uncontrollable transitions are included, and if no such transition leads to a marked state then at least one controllable transition with this property is included too. Thus, the result is a valid non-blocking supervisor depicted in Figure 5.2, equivalent to the supervisor shown in Figure 2.1.d.

The following results enunciate the correctness of DCS with respect to compositional supervisory control problems. In Theorem 3, we start by considering the soundness of the approach.

Theorem 3. Let $\mathcal{E} = (E, A_C)$ be a compositional supervisory control problem as per Definition 3, and let $\sigma : A_E^* \mapsto 2^{A_E}$ be the solution obtained by DCS for $\mathcal{E}$, then σ is a supervisor for $\mathcal{E}$.

Proof. The proof follows from the fact that for DCS to return a supervisor, the initial state $\bar{e}$ has to be flagged as a goal. Thus, there must be a marked CCC $\mathbb{C}$ containing $\bar{e}$. In such a case, it is straightforward to see that we can extract a supervisor from $\mathbb{C}$ (i.e., restricting the plant to remain within $\mathbb{C}$).

Let us assume that σ is not a valid supervisor for $\mathcal{E}$. Then σ must not fulfill one of the two solution requirements:

- Controllable: for this not to hold σ must disable some uncontrollable event from a state in which the event is enabled. Yet, in order to propagate a goal from an uncontrollable state e, DCS requires one successor state confirmed to lead to a goal, and all uncontrollable transitions con-

firmed not to lead to errors (i.e., e belongs to a CCC). Only when this happens, the state's status is propagated back to its ancestors, keeping all its outgoing uncontrollable transitions. Thus, there is no way for σ to ignore enabled uncontrollable events.

- Non-blocking: for this not to hold, there must exist a word $w \in \mathcal{L}^{\sigma}(E)$ such that w cannot be extended to reach a marked state. In other words, there is no $w' \in A_E^*$ such that $ww' \in \mathcal{L}^{\sigma}(E)$ and $\bar{e} \xrightarrow{ww'}_E e_m$ with $e_m \in M_E$. Thus, w can fall in one of two cases: (a) w reaches a deadlock; or (b) w closes a loop in a component $\mathbb{C}$ from which a marked state is not reachable. In the former, DCS would propagate an error upon expanding the deadlock state, making the state unreachable and contradicting the fact that $w \in \mathcal{L}^{\sigma}(E)$. In the latter, DCS would eventually explore every unconfirmed event in $\mathbb{C}$, which having exhausted the exploration options would empty the *open* queue. Thus, making the state unreachable and contradicting the fact that $w \in \mathcal{L}^{\sigma}(E)$. That is, in none of the two cases a goal could be propagated, since the resulting supervisor σ would not accept w.

Therefore, the assumption that the supervisor generated by the algorithm is not valid for E is absurd. □

In Theorem 4 we consider the completeness of the algorithm.

Theorem 4. Let $\mathcal{E} = (E, A_C)$ be a compositional supervisory control problem as per Definition 3. If there exists a solution for $\mathcal{E}$ then DCS returns a supervisor for $\mathcal{E}$.

Proof. The proof follows from the facts that: (a) the existence of a supervisor implies that a marked CCC containing the initial state exists (i.e., considering the states reachable by the supervisor); and (b) all outgoing transitions from a state e, considered during the on-the-fly exploration, are examined one by one. Still, if at some point we can conclude that an error can be uncontrollably induced from e, then e is added to a set of *Errors* and its remaining unexplored transitions are ignored. Similarly, if at some point we can conclude that a marked CCC can be controllably reached from e, then e is added to a set of *Goals* and its remaining unexplored transitions are also ignored.

Simply put, DCS progresses preserving the following invariants: (a) the *Errors* set is extended with states that should not be reached by a solution; (b) the *Goals* set is extended in such a way that it is kept a marked CCC; and (c) explored states have their transitions analyzed until exhausted or flagged as *Errors* or *Goals*. In worst case, DCS cannot flag states – as *Errors* or *Goals* – before exploring all their transitions (i.e., effectively computing the whole parallel composition). Given the problem has a solution, exploring the last transition must reveal a marked CCC containing the initial state.

We can prove this by induction on the exploration structure. Let us assume that there is a solution for $\mathcal{E}$. Then, by observing the algorithm in Listing 5.1 we can distinguish the following cases when expanding a state e' from a state e through an event ℓ:

(1) State e' has already been flagged as an error (e.g., a deadlock). Thus, a supervisor needs to prevent reaching e', and hence DCS will attempt to propagate an error. The propagation is interrupted immediately if ℓ is controllable and e has unconfirmed recommendations, since there could still be a valid option from e, and hence e is placed in the *open* queue. Otherwise, e is flagged as an error and the propagation continues since e does not lead to a marked state.

(2) State e' has already been flagged as a goal, that is, there is a marked CCC $\mathbb{C}$ containing e'. Thus, there is a word w that reaches a marked state e_m from e', and by prefixing ℓ to w we find a path from e to e_m. If e is "controllably closed" with *Goals*, the propagation of a goal ensues, otherwise, DCS propagates the undetermined status. In both cases the propagation returns a set of ancestor states with unconfirmed transitions that DCS will reopen. That is, the *open* queue does not become empty unless all ancestor states have been either explored exhaustively or flagged as goals.

(3) State e' is an ancestor of e (i.e., closing a loop) and there is a marked CCC $\mathbb{C}$ unfolding from e' and containing e. Thus, the plant cannot (uncontrollably) force an error from e and there exists at least one path reaching a marked state (i.e., a word passing through e can always be extended to reach a marked state). In such a case, DCS flags the

states of $\mathbb{C}$ as goals and propagates this fact backwards, which collects ancestor states to reopen.

(4) State e' is an ancestor of e (i.e., closing a loop), but there is no marked CCC $\mathbb{C}$ unfolding from e' and containing e. Thus, a word w reaching e cannot be extended to a word ww' that reaches a marked state of $\mathbb{C}$. In such a case, DCS propagates the undetermined status, which is interrupted upon considering an ancestor state with unconfirmed uncontrollable recommendations. The intermediate states covered during the propagation are then placed in the *open* queue. This is done because expanding the exploration structure further may reveal a super-set of the states in $\mathbb{C}$ with the required property.

(5) State e' is not flagged as a goal or an error and is not an ancestor of e. In this case e' is placed in the *open* queue, and the exploration continues until the solution is found.

Therefore, the *open* queue does not become empty as long as there are valid exploration options, and hence the complete state space is explored in worst case. Since there is a solution for $\mathcal{E}$, exploring the state space eventually leads to discover a marked CCC $\mathbb{C}$. In such event DCS flags the states in $\mathbb{C}$ as *Goals* and propagates this information backwards to ancestor states. The propagation iteratively extends the *Goals* set, and eventually reaches and flags the initial state. When this happens the exploration ends, and we build a supervisor by following transitions connecting states in *Goals*. □

5.2 Monotonic Abstraction

In this section, we present the Monotonic Abstraction (MA) and the heuristic function it induces. The MA has been previously proposed in the field of directed model-checking as an adaptation of a classic planning heuristic [50]. Despite the similarities between directed model-checking and DCS, the application of the MA to supervisory control requires additional considerations, namely dealing with controllable and non-blocking requirements.

The aim of the heuristic is to estimate the distance from a state to a goal without computing the parallel composition. However, in order to provide

informative estimates, the effects of synchronization need to be taken into account to some degree. To achieve this the MA behaves as if, after a transition, not only a new state is covered but also the source state is not abandoned. That is, the abstraction never drops a state once it reaches a point where the state is covered. This makes the abstraction consider a monotonically increasing set of covered states, and this is what gives it its name.

Definition 8 (Monotonic Abstraction)**.** Let $E = \{E_0, \ldots, E_n\}$ be a set of automata; $e = \langle e^0, \ldots, e^n \rangle$ a state of the parallel composition of E (i.e., $\forall 0 \leq j \leq n \;.\; e^j \in S_{E_j}$). The *Monotonic Abstraction* of E from e (denoted MA_E^e) is a maximal sequence $\langle s_0, a_0 \rangle, \ldots, \langle s_z, a_z \rangle$ of distinct pairs, with s_i a set of states in $2^{\bigcup_{j=0}^{n} S_{E_j}}$ and a_i a set of events in 2^{A_E} for all $0 \leq i \leq z$; such that:

- $s_0 = \{e^0, \ldots, e^n\}$, the initial set of states containing only the individual states of e;
- $a_i = \{\ell \mid \forall 0 \leq j \leq n \;.\; \ell \in A_{E_j} \Rightarrow (\exists t^j \in s_i \cap S_{E_j} \;.\; t^j \xrightarrow{\ell}_{E_j} t')\}$, the set of all enabled events, satisfying the rules of the parallel composition (Definition 2), from a state $\langle t^0, \ldots, t^n \rangle$ formed with the individual states t^j in s_i; and
- $s_i = s_{i-1} \cup \{t' \mid \exists \ell \in a_{i-1} \;.\; \exists 0 \leq j \leq n \;.\; \exists t^j \in s_{i-1} \cap S_{E_j} \;.\; t^j \xrightarrow{\ell}_{E_j} t'\}$, the set of all states already in s_{i-1}, plus the states reachable through a step labeled with an event from a_{i-1}, and originating from a state $\langle t^0, \ldots, t^n \rangle$ formed with states in s_{i-1}.

The abstraction is a sequence of sets of the original states and events. Each state in a set s_i belongs to the joint set of states of all the automata in E. Each set a_i contains the enabled events from any state of E that can be formed with states in s_i. The states in s_i are the states in s_{i-1} plus all new states reachable from states of s_{i-1} through events in a_{i-1}. Thus, states are collected as they become reachable from previously covered states, but only if they can be reached taking the rules of synchronization into account. The last element of the sequence $\langle s_z, a_z \rangle$ is such that no new states can be reached from states in s_z through events in a_z.

At first glance, by considering the power set of states and events, the abstraction may seem to incur in an exponential blowup. Yet, since the abstraction

applies a rule of monotonic growth, the reachable sets of states are restricted to sets that contain all the traversed states from the initial state. Thus, the abstraction size is polynomial with respect to the number of states in the intervening components. Furthermore, the MA obviates the computation of the parallel composition, since it does not check whether states are reachable in the composition (i.e., it only checks if transitions synchronize).

Property 1. Let $MA^{e}_{E} = \langle s_0, a_0\rangle, \ldots, \langle s_z, a_z\rangle$ be the MA of the set of automata $E = \{E_0, \ldots, E_n\}$ from a state $e = \langle e^0, \ldots, e^n\rangle$ of E. The length of MA^{e}_{E} is, in worst case, $\sum_{i=0}^{n} |S_{E_i}|$.

Proof. The proof is straightforward since once every state in $\bigcup_{i=0}^{n} S_{E_i}$ is included in a set of states s, no other distinct set of states can be reached. Thus, in worst case, each step contributes only one fresh state, leading to $\sum_{i=0}^{n} |S_{E_i}|$ steps before reaching the last possible fresh state. □

Despite the fact that they look very dissimilar, there is a strong relation between a model and its abstraction. In particular, for every run in the composed automaton E starting from the initial state $\bar{e}$, there is a path in the abstraction. In other words, if $E = \{E_0, \ldots, E_n\}$ is a set of automata; $MA^{e_0}_{E} = \langle s_0, a_0\rangle, \ldots, \langle s_z, a_z\rangle$ is the MA of E from the initial state $\bar{e} = e_0$; and $e_0 \xrightarrow{w}\!\!\!\!\cdots\!\!\!\!\rightarrow_E e_k$ is a run on the composed automaton E over word $w = \ell_0, \ldots, \ell_k$. Then, for all $0 \leq i \leq z$ it holds that $\ell_i \in a_i$ and $e_i \in s_i$; while for all $z < j \leq k$, $\ell_j \in a_z$ and $e_j \in s_j$. However, the converse is not necessarily true.

Definition 9 (MA Path)**.** Let $MA^{e}_{E} = \langle s_0, a_0\rangle, \ldots, \langle s_z, a_z\rangle$ be the MA of the set of automata $E = \{E_0, \ldots, E_n\}$ from a state $e = \langle e^0, \ldots, e^n\rangle$ of E. A *path* in MA^{e}_{E} is a sequence of tuples of states and events $\langle e_0, \ell_0\rangle, \ldots, \langle e_k, \ell_k\rangle$, such that for every $0 \leq i \leq k$ it holds that $e_i \in s_z$, $\ell_i \in a_z$ and $e_i \xrightarrow{\ell}_{E_j} e_{i+1}$ is a valid step for some $E_j \in E$. In other words, a path represents a sequence of connected steps considering states and events reachable in MA^{e}_{E}.

The heuristic function estimates the distance to the goal by considering paths starting at the initial state and reaching a marked state. However, in order to provide a good approximation, the heuristic also needs to take into account the moment at which each state and event in the path were included in the MA, we call this the state's (or event's) "generation."

Definition 10 (MA Generation)**.** Let $MA^e_E = \langle s_0, a_0\rangle, \ldots, \langle s_z, a_z\rangle$ be the MA of the set of automata $E = \{E_0, \ldots, E_n\}$ from a state $e = \langle e^0, \ldots, e^n\rangle$ of E. We define the *generation* of a state $t \in s_z$ (or the *generation* of an event $t \in a_z$) as the index of the sequence in which t appears for the first time, denoted $g(t)$. More formally,

$$g(t) = i, \text{ where } i \text{ is such that: } t \in s_i \cup a_i \wedge \forall 0 \leq j < i \;.\; t \notin s_j \cup a_j$$

Note that, with a slight abuse of notation, we define the generation for states and events simultaneously. The generation is useful since it allows to detect whether an event requires intermediate synchronization steps before being enabled from a given state. That is, if there is a difference between the generations of a state e and an event ℓ (i.e., $g(\ell) - g(e) > 0$), we can deduce that intermediate steps will be required before being able to take a step from e through ℓ. This allows the heuristic to obtain a better approximation of the distance to the goal, than what would be obtained by naively taking the length of a path.

Definition 11 (MA Distance)**.** Let $MA^e_E = \langle s_0, a_0\rangle, \ldots, \langle s_z, a_z\rangle$ be the MA of the set of automata $E = \{E_0, \ldots, E_n\}$ from a state $e = \langle e^0, \ldots, e^n\rangle$ of E. We define the *distance* of a path $p = \langle e_0, \ell_0\rangle, \ldots, \langle e_k, \ell_k\rangle$ in MA^e_E, denoted $|p|$, as the sum of the relative distances of each ℓ_i with respect to e_i for $0 \leq i \leq k$, also denoted $|\langle e_i, \ell_i\rangle|$. More formally,

$$\begin{aligned} |p| &= \textstyle\sum_{i=0}^{k} |\langle e_i, \ell_i\rangle| \\ |\langle e, \ell\rangle| &= 1 + \max\{0, g(\ell) - g(e)\} \end{aligned}$$

Since reaching a goal requires all automata to reach marked states simultaneously, we need to consider the distance of paths reaching marked states for every automaton. Furthermore, our goal is not simply to reach a marked state, but to always be able to extend words to reach such a state. Thus, in order to direct the exploration towards this goal it is insufficient to estimate the distance to an isolated marked state. Instead, we prioritize reaching an already visited marked state, in an attempt to guide the exploration into closing a loop. The information of the already visited marked states can be easily

extracted from the exploration structure kept by the on-the-fly exploration procedure. Therefore, we want the heuristic to prioritize events leading to already visited marked states, while relegating to a second place events leading to non-visited marked states.

Definition 12 (MA Heuristic Estimate)**.** Let $MA^e_E = \langle s_0, a_0\rangle, \ldots, \langle s_z, a_z\rangle$ be the MA of a set of automata $E = \{E_0, \ldots, E_n\}$ from a state $e = \langle e^0, \ldots, e^n\rangle$ of E; and let $\mathcal{M} \subseteq \bigcup_{i=0}^{n} M_{E_i}$ be a set of (already visited) marked states. The *heuristic estimate* of an event ℓ in MA^e_E (denoted $MA^e_E(\ell)$) is a tuple $\langle m, d\rangle \in \{0,1\} \times \mathbb{N}$, such that $\langle m, d\rangle = \max\{\langle m_0, d_0\rangle, \ldots, \langle m_n, d_n\rangle\}$, where tuples are compared lexicographically and for each $0 \leq i \leq n$ it holds the following:

- if $\ell \in A_{E_i}$:
  - $m_i = 0$ and $d_i = |r|$, if r is the shortest path in E_i starting with ℓ from e^i and reaching $t^i \in M_{E_i} \cap \mathcal{M}$;
  - $m_i = 1$ and $d_i = |r|$, if r is the shortest path in E_i starting with ℓ from e^i and reaching $t^i \in M_{E_i} \setminus \mathcal{M}$, and there is no path starting with ℓ from e^i reaching a state in $M_{E_i} \cap \mathcal{M}$;
  - $m_i = 1$ and $d_i = \infty$, if there is no path in E_i starting with ℓ from e^i and reaching a state in M_{E_i}; or
- if $\ell \notin A_{E_i}$:
  - $m_i = 0$ and $d_i = 0$, the minimum possible value.

This definition of heuristic estimate considers whether an event ℓ is expected to lead to an already visited marked state ($m = 0$), or a non-visited marked state ($m = 1$). The heuristic estimate considers that ℓ leads to an already visited marked state only when it does so in every intervening component (otherwise taking the maximum causes m to be 1). Additionally, the heuristic estimate approximates the number of steps required to achieve the goal with value d, which can be ∞ when marked states are estimated to be unreachable through ℓ in at least one automaton.

Note that when ℓ does not belong to the event set of an automaton E_i (i.e., $\ell \notin A_{E_i}$), its occurrence does not change the current state of E_i, and hence it does not affect the estimated distance to a state of M_{E_i}. For this reason, in

such a case we set the estimate to the minimum possible value (i.e., $\langle 0, 0 \rangle$). This value is discarded when taking the maximum with respect to the distance of ℓ in an automaton E_j with ℓ in its event set (i.e., $\ell \in A_{E_j}$).

Example 6. *Recall the example from Figure 2.1 where two automata C and F, represent the interaction between a customer and a factory. The $MA^{\langle c_0, f_0 \rangle}_{\{C,F\}}$ is depicted in Figure 5.3.a, indexed with the generation at which each state and event was included in the abstraction. Note that we highlight in bold the fresh elements of each generation.*

$$
\begin{aligned}
0: \ & \langle \{\boldsymbol{c_0}, \boldsymbol{f_0}\}, && \{\boldsymbol{p_1}, \boldsymbol{p_2}, \boldsymbol{r_1}, \boldsymbol{r_2}\} \rangle \\
1: \ & \langle \{c_0, \boldsymbol{c_1}, \boldsymbol{c_2}, f_0, \boldsymbol{f_1}, \boldsymbol{f_2}\}, && \{\boldsymbol{d_1}, \boldsymbol{d_2}, p_1, p_2, r_1, r_2\} \rangle
\end{aligned}
$$

(a) $MA^{\langle c_0, f_0 \rangle}_{\{C,F\}}$

$$
\begin{aligned}
0: \ & \langle \{\boldsymbol{c_1}, \boldsymbol{f_0}\}, && \{\boldsymbol{p_1}, \boldsymbol{p_2}\} \rangle \\
1: \ & \langle \{c_1, f_0, \boldsymbol{f_1}, \boldsymbol{f_2}\}, && \{\boldsymbol{d_1}, p_1, p_2\} \rangle \\
2: \ & \langle \{\boldsymbol{c_0}, c_1, f_0, f_1, f_2\}, && \{d_1, p_1, p_2, \boldsymbol{r_1}, \boldsymbol{r_2}\} \rangle \\
3: \ & \langle \{c_0, c_1, \boldsymbol{c_2}, f_0, f_1, f_2\}, && \{d_1, \boldsymbol{d_2}, p_1, p_2, r_1, r_2\} \rangle
\end{aligned}
$$

(b) $MA^{\langle c_1, f_0 \rangle}_{\{C,F\}}$

Figure 5.3. MA of automata C and F.

In this example we can see that the MA rapidly collects all reachable states. The first element of the sequence contains a set of states with only the initial states (i.e., $\{c_0, f_0\}$) and the set of enabled events (i.e., $\{p_1, p_2, r_1, r_2\}$). The second element of the sequence also contains the states reachable in the individual components after following a step with one of the enabled events. Namely,

- *c_1 due to $c_0 \xrightarrow{r_1}_C c_1$;*
- *c_2 due to $c_0 \xrightarrow{r_2}_C c_2$;*
- *f_1 due to $f_0 \xrightarrow{p_1}_F f_1$; and*
- *f_1 due to $f_0 \xrightarrow{p_2}_F f_2$.*

With the states in the resulting set we can form states $\langle c_1, f_1 \rangle$ and $\langle c_2, f_2 \rangle$, among others. Shared events d_1 and d_2 are enabled in $\langle c_1, f_1 \rangle$ and $\langle c_2, f_2 \rangle$

respectively, and hence they are present in the second set of enabled events. And, since we have already covered all the states of S_E at this point, it is clear that we have found the last element of the abstraction.

DCS *starts the on-the-fly exploration at state $\langle c_0, f_0\rangle$, which is marked since c_0 and f_0 are marked, and hence we consider these states as "visited". Thus, the heuristic should prioritize events leading back to these states (i.e., in an attempt to guide the exploration into closing a loop).*

The enabled events from $\langle c_0, f_0\rangle$ are $\{p_1, p_2, r_1, r_2\}$. Since $r_1 \notin A_F$ and $r_2 \notin A_F$, the heuristic estimates for r_1 and r_2 with respect to F are both $\langle 0, 0\rangle$. Yet, since $r_1 \in A_C$ and $r_2 \in A_C$, we need to look for the shortest paths in C starting with r_1 and r_2 going from c_0 back to c_0, namely $c_0 \xrightarrow{r_1}_C c_1 \xrightarrow{d_1}_C c_0$ and $c_0 \xrightarrow{r_2}_C c_2 \xrightarrow{d_2}_C c_0$. Both paths have a distance of $\langle 0, 2\rangle$ (i.e., each step has a distance of 1). Therefore, the heuristic estimates for r_1 and r_2 are $\langle 0, 2\rangle$ (i.e. $\max\{\langle 0, 0\rangle, \langle 0, 2\rangle\}$). The analysis for p_1 and p_2 is similar, but since these are controllable, the heuristic function will prioritize r_1 and r_2.

Given that the estimates are equal, the abstraction sets no preference in the exploration order between events r_1 and r_2. Let us assume that DCS *first explores event r_1 expanding state $\langle c_1, f_0\rangle$ at iteration 1. In Figure 5.4, we depict the complete exploration structure built by* DCS *indexing each state with the number of the iteration in which it was expanded. At this point, only states $\langle c_0, f_0\rangle_0$ and $\langle c_1, f_0\rangle_1$ have been expanded. In Figure 5.3.b, we depict the MA of C and F from state $\langle c_1, f_0\rangle$. Note that shared event d_1 is included at generation 1 while d_2 is included only at generation 3. This is because, for d_2 to be enabled, we need states c_2 and f_2, and c_2 is only covered at generation 3.*

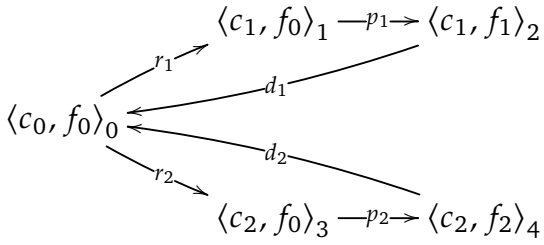


Figure 5.4. Exploration structure built guided by the MA.

In this case, the enabled events are p_1 and p_2 (i.e., $\langle c_1, f_0\rangle$ is a fully control-

lable state), which are only in the event set of F. The paths leading back to f_0 starting with these events are:

- $f_0 \xrightarrow{p_1}_F f_1 \xrightarrow{d_1}_F f_0$, *with a distance of* $\langle 0, 2\rangle$ *since both steps have a distance of 1; and*
- $f_0 \xrightarrow{p_2}_F f_2 \xrightarrow{d_2}_F f_0$, *with a distance of* $\langle 0, 4\rangle$ *since the first step has a distance of 1 and the second step has a distance of 3 (*$g(d_2) = 3$ *and* $g(f_2) = 1$*).*

Therefore, DCS *will continue exploring p_1, expanding state $\langle c_1, f_1\rangle$ at iteration 2. From $\langle c_1, f_1\rangle$ the only enabled event is d_1, which closes a loop over $\langle c_0, f_0\rangle$. No CCC unfolds from $\langle c_0, f_0\rangle$ since the uncontrollable event r_2 remains unexplored from that state. Thus, state $\langle c_0, f_0\rangle$ is reopened.*

In the following, DCS *will proceed as in the previous case, but exploring events r_2 at iteration 3, p_2 at iteration 4 and d_2 at iteration 5. After closing another loop over $\langle c_0, f_0\rangle$, a CCC is discovered and the propagation procedure will flag the explored states as goals. This time $\langle c_0, f_0\rangle$ will not have remaining uncontrollable options unexplored, and hence the propagation will flag it as a goal. At this point, the exploration will stop and a supervisor will be constructed by following the states flagged as goals from the initial state. The supervisor obtained from the exploration structure is equivalent to the one depicted in Figure 2.1.d.*

5.2.1 Efficient Computation of the Monotonic Abstraction

Given that the heuristic function has to be computed for every expanded state, it is important to compute the abstraction efficiently. A naive approach, that is, building the abstraction as per Definition 8 and computing the estimate for every path, would be unnecessarily costly. Instead, we show how to efficiently compute the shortest paths for each automaton taking the difference in generations into account.

In Listing 5.4 we show how to build the MA given a set of automata and a

```
procedure buildMA(E = {E_0, ..., E_n}, e = ⟨e^0, ..., e^n⟩):
  fresh = {e^0, ..., e^n}
  iteration = 0
  update(fresh, iteration)
  while fresh ≠ ∅:
    iteration = iteration + 1
    fresh = ∅
    for ℓ ∈ generations:
      for e ∈ pending[ℓ]:
        if e ∉ generations:
          fresh = fresh ∪ {e}
      pending[ℓ] = ∅
    update(fresh, iteration)
  return generations // (i.e., MA^e_E)

procedure update(fresh, iteration):
  for e ∈ fresh:
    generations[s] = iteration
    for (e, ℓ, e') ∈→_{E_i} such that e ∈ S_{E_i}:
      pending[ℓ] = pending[ℓ] ∪ {e'}
      ready[E_i] = ready[E_i] ∪ {ℓ}
      if ℓ ∉ generations ∧ isEnabled(ℓ)
        generations[ℓ] = iteration

function isEnabled(ℓ):
  return ∀0 ≤ i ≤ n . ℓ ∈ A_{E_i} ⇒ ℓ ∈ ready[E_i]
```

Listing 5.4. MA building procedure.

state. For ease of presentation we use the following (initially empty) "global" tables:

- *generations*, a table that maps states and events to their generation in the abstraction;
- *ready*, a table that, for each automaton E_i, holds the events ready from an already covered state of E_i;
- *pending*, a table that, for every label ℓ, holds the states reachable through ℓ from already covered states;

At each iteration a new element of the MA sequence is built. In worst case, each iteration only adds one *fresh* state e_i from some automaton E_j in E. A state e_i added at the i-th iteration has *generation* i (i.e., $g(e_i) = i$). For each step $e_i \xrightarrow{\ell}_{E_j} e'$, we add an entry on the table holding *pending* steps, and we set ℓ as *ready* in E_j. We also check if an event ℓ became "enabled" after

```
procedure evaluateMA(MA^e_E, M):
  estimates = ∅
  for ℓ ∈ generations such that generations[ℓ] = 0: // (i.e., ℓ enabled)
    target = ⋃_{i=0}^{n} M_{E_i}
    if select(ℓ, target) ≠ {i | 0 ≤ i ≤ n ∧ ℓ ∈ A_{E_i}}:
      estimates[ℓ] = ⟨1, ∞⟩
    else:
      selected = select(ℓ, M)
      if selected = {i | 0 ≤ i ≤ n ∧ E_i ∈ M ∧ ℓ ∈ A_{E_i}}:
        for i ∈ selected:
          estimates[ℓ] = max{estimates[ℓ], ⟨0, minPath(e^i, ℓ, M)⟩}
      else:
        for 0 ≤ i ≤ n such that i ∉ selected:
          estimates[ℓ] = max{estimates[ℓ], ⟨1, minPath(e^i, ℓ, target)⟩}
  return rank(estimates)

function select(ℓ, target):
  return {i | ∃t ∈ target . t ∈ strides[e^i][ℓ]}

function minPath(e^i, ℓ, target):
  res = 0
  for (e^i, ℓ, e') ∈ →_{E_i}:
    res = max{res, |⟨e^i, ℓ⟩| + A*(generations, strides, e', target)}
  return res
```

Listing 5.5. MA heuristic evaluation.

setting it as ready in E_j (i.e., $g(\ell) = i$). Then, for each enabled event ℓ, we consume the *pending* steps through ℓ, potentially discovering *fresh* states to consider in the following iteration. This process is repeated until no *fresh* states are discovered. Finally, we return the *generations* of all reachable states and events, which we use for the computation of heuristic estimates.

In worst case, building the MA has a complexity of $O(\sum_{i=0}^{n} |\rightarrow_{E_i}|)$, since we may consider every transition for each automaton E_i of E.

In Listing 5.5, we show how to efficiently extract heuristic estimates for enabled events given a MA and a set of (already visited) marked states.

The procedure uses a precomputed table (named *strides*) with the number of steps between any two states for each automaton E_i in E. The *strides* table is initialized once and reused at every invocation of the MA, and hence its cost is amortized during the exploration. The repeated Dijkstra's algorithm [75] is used for each state of E_i, obtaining the events leading from one state to another, and also the number of steps between them. Note that the number

of steps between states of E_i does not take into account the effects of synchronization. Thus, the number of steps always underestimates the distance of a path p connecting two states, since each step of p has a distance greater or equal to 1. In other words, the distance of p is always greater or equal to the number of steps in p.

For every enabled event ℓ we look for the shortest paths reaching marked states of each automaton in E. In the case that there is at least one automaton incapable of reaching a single marked state, we default the heuristic estimate to $\langle 1, \infty \rangle$ (i.e., skipping the computation of paths for every automata). Otherwise, we start by selecting each E_i with no reachable states in $M_{E_i} \cap \mathcal{M}$, for the computation of paths. Since we are interested in the farthest marked state of E, once we find an automaton incapable of reaching a state of $\mathcal{M}$, we can be sure that its shortest path to a marked state will be longer than one belonging to an automaton reaching a state in $\mathcal{M}$. That is, when at least one automata cannot reach a state of $\mathcal{M}$, we can then skip the computation of paths for automata that do reach states in $\mathcal{M}$. Only when all automata in E can reach a state of $\mathcal{M}$, we select them all.

For each selected automaton E_i we run the A* [49] procedure from the states reachable after a step through ℓ from the initial state (i.e., a single state for deterministic automata). This will search for the path with minimum distance between the current state and a target state of E_i (i.e., a state of $\mathcal{M}_{E_i} \cap \mathcal{M}$ or $\mathcal{M}_{E_i} \setminus \mathcal{M}$ depending on the selected automata). We guide A* with the precomputed number of steps between states of E_i. The accumulated cost of a path explored by A* is the sum of distances of the steps in the path. Since the number of steps underestimates the distance, A* is guaranteed to return the shortest path to a target state of E_i. Interestingly, we can avoid repeating the work performed by A* by caching the paths explored for each state. That is, in worst case, A* needs to consider every state only once.

Finally, we take the maximum between the distances of the paths considered (or $\langle 1, \infty \rangle$ if no automata were selected), and use that value as the heuristic estimate for event ℓ. The ranking is generated by the `rank` function, which sorts uncontrollable events in descending order and then appends controllable events sorted in ascending order.

Again, extracting heuristic estimates from the MA has a worst case complexity of $O(\sum_{i=0}^{n} |{\rightarrow_{E_i}}|)$. This is because we may consider every transition for each automaton E_i of E, and that we can amortize different invocations to A* by caching intermediate results.

5.3 Ready Abstraction

In this section we present the Ready Abstraction (RA) and the heuristic function it induces. The RA represents a reduced version of the state space which disregards enough information to avoid the state explosion problem, but at the same time preserves enough details to allow extracting useful heuristic estimates.

The RA only considers the events that are ready in the individual states of a composed state, that is, the events available from the current state of each component. We focus on ready events because they allow to make a local estimate of the distance to a goal state in each individual automaton. When a non-enabled ready event ℓ is estimated to move the exploration closer to a goal, we recursively look for ready events that move the exploration towards a state where ℓ becomes enabled. That is, by computing the RA we build a dependencies graph that indicates which ready events move the exploration towards marked states, and also which other ready events need to occur before them.

Definition 13 (Ready Abstraction)**.** Let $E = \{E_0, \ldots, E_n\}$ be a set of automata, and $e = \langle e^0, \ldots, e^n \rangle$ a state of the parallel composition of E (i.e., $\forall 0 \leq i \leq n \;.\; e^i \in S_{E_i}$). The *Ready Abstraction* of E from e (denoted RA_E^e) is a directed graph (V, D), where:

- $V = \{\ell \mid \exists 0 \leq i \leq n \;.\; e^i \xrightarrow{\ell}_{E_i} e^{i\prime}\}$, is a set of vertices containing the ready events in e; and
- $D = \{(\ell, \ell') \mid \exists 0 \leq i \leq n \;.\; e^i \xrightarrow{\ell \ldots \ell'}_{E_i} e^{i\prime}\}$, is a set of edges (ordered pairs) connecting vertices if there exists a run between them in some automaton E_i of E.

Where $e^i \xrightarrow{\ell \ldots \ell'}_{E_i} e^{i\prime}$ denotes a run (c.f., Notation 1) from state $e^i \in S_{E_i}$ (i.e.,

the state of automaton E_i in e) through a word $w = \ell, \ldots, \ell'$ and reaching a state $e^{i'} \in S_{E_i}$.

Remarkably, the RA is a graph of polynomial size with respect to the number of event labels in a compositional description. In particular, the RA is polynomial with respect to the number of ready events, which is typically small compared to the number of transitions, and hence it can be built very efficiently.

Property 2. Let $RA_E^e = (V, D)$ be a directed graph representing the RA of the set of automata $E = \{E_0, \ldots, E_n\}$ from state $e = \langle e^0, \ldots, e^n \rangle$ of E. In worst case, it happens that:

- $|V| = \sum_{i=0}^{n} |A_{E_i}|$; and
- $|D| \leq |V|^2$.

Proof. The proof is straightforward since: (a) in worst case, all the events in $\bigcup_{i=0}^{n} A_{E_i}$ are ready from e; and (b) the edges may connect every pair of ready events. □

Observe that a path in the RA graph represents a casual relation between events. That is, a path represents a series of consecutive runs (with the last event of a run equal to the first event of the following run) in potentially different automata of E.

Definition 14 (RA Path)**.** Let $RA_E^e = (V, D)$ be a directed graph representing the RA of the set of automata $E = \{E_0, \ldots, E_n\}$ from state $e = \langle e^0, \ldots, e^n \rangle$ of E. A *path* in RA_E^e is a sequence of vertices $\ell_0, \ldots, \ell_m$, such that $(\ell_i, \ell_{i+1}) \in D$ for all $0 \leq i < m$. As a consequence of being a RA, each edge (ℓ_i, ℓ_{i+1}) in the graph entails the existence of a run $e^j \xrightarrow{\ell_i \ldots \ell_{i+1}}_{E_j} e^{j'}$ for some $0 \leq j \leq n$.

In particular, we are interested in paths leading to marked states. Such paths suggest an orderly sequence of steps required to reach a marked state. Of course, since the abstraction only considers ready events, the path may be incomplete.

Definition 15 (RA Marked Path)**.** We call *marked path* to a path in a RA ending in a marked state. That is, a path $p = \ell_0, \ldots, \ell_m$, is "marked" if and

only if:

$$\exists 0 \leq j \leq n \;.\; e^j \xrightarrow{\ell_m \ldots \ell'}_{E_j} e^{j'} \wedge e^{j'} \in M_{E_j}$$

Observe that $(\ell_i, \ell_{i+1}) \in D$ for every $0 \leq i < m$, since p is a path. And also note that event ℓ', leading to marked state $e^{j'}$, is not part of the path p as it may not be ready (i.e., it might be the case that $\ell' \notin V$).

In order to provide an estimated distance to the goal, we consider all marked paths in the RA that start with enabled events (i.e., ready in all the intervening components). Such a path indicates that it might be possible to reach a marked state by following the events in the path in order, starting with the event currently enabled. Then, the distance to the goal can be approximated by the number of steps of each run in a path going from an enabled event to a marked state.

Definition 16 (RA Distance)**.** Let $RA^e_E = (V, D)$ be a directed graph representing the RA of the set of automata $E = \{E_0, \ldots, E_n\}$ from state a $e = \langle e^0, \ldots, e^n \rangle$ of E; and let $p = \ell_0, \ldots, \ell_m$ a path of RA^e_E. Since p is a path of RA^e_E, for all $0 \leq i \leq m$ there must be at least an automaton E_k in E containing a run $r^k_i = e^k \xrightarrow{\ell_i \ldots \ell_{i+1}}_{E_k} e^{k'}$ with minimum length (i.e., $|r^k_i|$ is the minimum number of steps in E_k starting with ℓ_i from e^k and ending with ℓ_{i+1}, or 0 if ℓ_i is not ready from e^k). We define the estimated *distance* of p (denoted $|p|$) as the sum of lengths of the greatest shortest runs contained in p, that is,

$$|p| = \sum_{i=0}^{m} \max_k |r^k_i|$$

As in the MA our goal is not simply to reach a marked state, but to always be able to extend a word to reach such a state. Thus, we prioritize paths in the abstraction reaching already visited marked states, in an attempt to guide the exploration into closing a loop. As before, we can easily extract this information from the exploration structure.

Definition 17 (RA Event Estimate)**.** Let RA^e_E be a directed graph representing the RA of the set of automata $E = \{E_0, \ldots, E_n\}$ from a state $e = \langle e^0, \ldots, e^n \rangle$ of E; and let $\mathcal{M} \subseteq \bigcup_{i=0}^{n} M_{E_i}$ be a set of (already visited) marked states. The heuristic estimate of an event ℓ in RA^e_E (denoted $RA^e_E(\ell)$) is a tuple

$\left\langle \langle m_0, d_0 \rangle, \ldots, \langle m_n, d_n \rangle \right\rangle \in (\{0,1\} \times \mathbb{N})^n$, where for each $0 \leq i \leq n$ the tuple $\langle m_i, d_i \rangle$ is:

- If there is a path starting with ℓ and reaching a state of M_{E_i}:
  - $\langle 0, |p| \rangle$ if p is the shortest path starting with ℓ and reaching a state in $M_{E_i} \cap \mathcal{M}$; or
  - $\langle 1, |p| \rangle$ if p is the shortest path starting with ℓ and reaching a state in $M_{E_i} \setminus \mathcal{M}$, and there is no path starting with ℓ and reaching a state in $M_{E_i} \cap \mathcal{M}$.
- If there is no path starting with ℓ and reaching a state of M_{E_i}:
  - $\langle 0, |p| + 1 \rangle$ if $\ell \notin A_{E_i}$ and p is the shortest path reaching a state in $M_{E_i} \cap \mathcal{M}$;
  - $\langle 1, |p| + 1 \rangle$ if $\ell \notin A_{E_i}$ and p is the shortest path reaching a state in $M_{E_i} \setminus \mathcal{M}$, and there is no path reaching a state in $M_{E_i} \cap \mathcal{M}$; or
  - $\langle 1, \infty \rangle$ if $\ell \in A_{E_i}$.

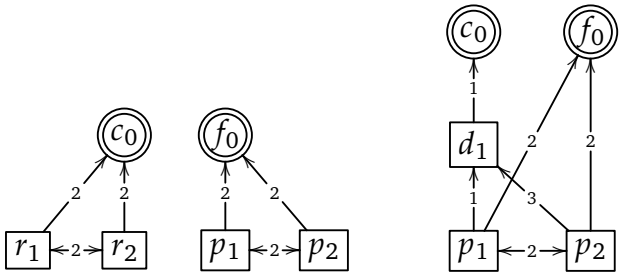
This definition of heuristic estimate considers for each automaton E_i of E, whether an event ℓ promises to reach an already visited marked state ($m_i = 0$), or a non-visited marked state ($m_i = 1$). Additionally, the heuristic estimate approximates the number of steps required to reach a target state in each automaton (with values d_i). When there is no marked path starting with ℓ the heuristic considers whether ℓ belongs to the event set of E_i or not. On the one hand, if $\ell \notin A_{E_i}$ then the event does not modify the current state of E_i. Hence, we simply consider the shortest distance to a marked state of E_i plus 1 to take the occurrence of ℓ into account. On the other hand, if $\ell \in A_{E_i}$ the fact that there is no marked path starting with ℓ, means that ℓ leads to a state where marked states of E_i are unreachable ($d_i = \infty$).

Observe that unlike the case of the MA, the heuristic estimates extracted from the RA are tuples instead of single values (i.e., an estimated distance for every automaton). This is because, the RA does not account for the number of intermediate synchronization steps that might be required to follow a given path. And hence, we need additional criteria to perform a fine grained comparison. While this approach could also be applied to the MA, it would not add much

information since it already considers (to some extent) the intermediate steps required in other automata.

In order to compare two estimates (i.e., sequences of tuples) we first sort them in descending order and then we apply the standard lexicographical comparison. Thus, this comparison first contrasts the distances of the automata where a target state is most distant and, only in case of a tie, continues comparing the estimates for the automata which are "increasingly closer" to their respective target states. In particular, when an event ℓ is estimated to reach an already visited marked state in every automata, the exploration of ℓ will be prioritized over the exploration of events leading to non-visited marked states in at least one automaton. Whereas, when the estimate for ℓ is $\langle 1, \infty \rangle$ for some automaton E_i, that is no marked state of E_i seems reachable through ℓ, this value will be considered first during lexicographical comparison (i.e., effectively relegating the exploration of ℓ).

Example 7. *Recall the example of Figure 2.1 where two automata, C and F, represented the interaction between a customer requesting one of two products and a factory capable of producing and delivering the products one at a time. The initial states c_0 and f_0 are marked, and hence we can find a supervisor for this example by finding a loop enclosing the state $\langle c_0, f_0 \rangle$.*



(a) $RA^{\langle c_0, f_0 \rangle}_{\{C,F\}}$ **(b)** $RA^{\langle c_1, f_0 \rangle}_{\{C,F\}}$

Figure 5.5. RA of automata C and F.

DCS *starts the exploration at state $\langle c_0, f_0 \rangle$ and computes the RA from this state. For illustrative purposes in Figure 5.5.a we depict the RA from $\langle c_0, f_0 \rangle$ extended with the information of reachable marked states. The ready events from $\langle c_0, f_0 \rangle$ are $\{p_1, p_2, r_1, r_2\}$, which in this particular case are all enabled.*

Events p_1 and p_2 are controllable while r_1 and r_2 are uncontrollable (i.e., a mixed state). The shortest path starting with r_1 and reaching c_0 has a distance of 2. Given that: (a) $r_1 \notin A_F$; (b) there is no path starting with r_1 and getting to a state in M_F; and (c) the shortest path from f_0 back to f_0 has a length of 2; the estimated distance of r_2 to f_0 is 3 (i.e., accounting for the occurrence of r_2). The abstraction is symmetric, giving equal estimates for r_1 and r_2 (i.e., $\langle\langle 0,2\rangle, \langle 0,3\rangle\rangle$). The analysis for p_1 and p_2 is analogous.

Let us consider that DCS *explores r_1 first, reaching state $\langle c_1, f_0\rangle$ at iteration 1. In Figure 5.6 we depict the complete exploration structure built by* DCS *indexing each state with the number of the iteration in which it was expanded. In Figure 5.5.b we depict the RA from state $\langle c_1, f_0\rangle$ again extended with reachable marked states. The ready events from this state are $\{d_1, p_1, p_2\}$, but only p_1 and p_2 are enabled (i.e., a fully controllable state). Starting with p_1 there are marked paths reaching c_0 and f_0 both with a distance of 2, hence the estimate for p_1 is $\langle\langle 0,2\rangle, \langle 0,2\rangle\rangle$. Starting with p_2 there are marked paths reaching c_0 and f_0 with distances of 4 and 2 respectively, hence the estimate for p_2 is $\langle\langle 0,4\rangle, \langle 0,2\rangle\rangle$. Therefore, the exploration will continue by exploring p_1 and expanding $\langle c_1, f_1\rangle$ at iteration 2.*

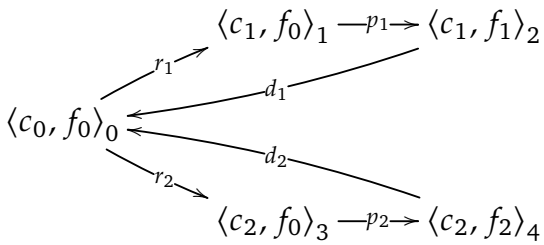


Figure 5.6. Exploration structure built guided by the RA.

The exploration will continue considering $\langle c_1, f_1\rangle$, whose only enabled event is d_1 reaching $\langle c_0, f_0\rangle$ and closing a loop with a marked state. From $\langle c_0, f_0\rangle$ the uncontrollable event r_2 is still unexplored, and hence no CCC unfolds from the state. Thus, state $\langle c_0, f_0\rangle$ is reopened.

In the following, event r_2 is explored from $\langle c_0, f_0\rangle$, expanding state $\langle c_2, f_0\rangle$ at iteration 3. Then, the exploration continues similarly as in the case for r_1, but exploring p_2 and expanding $\langle c_2, f_2\rangle$ at iteration 4. From $\langle c_2, f_2\rangle$, event d_2 is explored closing a loop over $\langle c_0, f_0\rangle$. This time a CCC containing the

explored states unfolds from $\langle c_0, f_0 \rangle$, and hence the propagation procedure flags these states as goals. That is, $\langle c_0, f_0 \rangle$ is flagged as a goal and the algorithm terminates successfully.

Coincidentally, the exploration structure is the same as in the MA case depicted in Figure 5.4. As before, it is straightforward to see that the supervisor, obtained by following the states flagged as goals, is equivalent to the one depicted in Figure 2.1.d.

5.3.1 Efficient Computation of the Ready Abstraction

As in the case of the MA it is important to compute the RA efficiently given that the heuristic is evaluated for every state. In Listing 5.6 we show an algorithm for building the RA. The algorithm assumes the existence of a precomputed table, called *strides*, indicating for every state of every automaton, which events lead to other states and in how many steps. Similarly to the evaluation of the MA, the *strides* table is precomputed once at an initialization step running the repeated Dijkstra's algorithm over each individual automaton. Thus, the cost of precomputing this table is amortized during the exploration.

Additionally, in this case, we enrich the *strides* table with the information of which events lead to other events from a given state, and in how many steps. That is, from the knowledge that a state e is reachable through an event ℓ in k steps, we deduce that ℓ leads to the ready events from e in k steps. The algorithm takes as input the compositional description of the plant and the current state of the exploration, and computes a graph by following Definition 13 closely.

```
procedure buildRA(E =_ℓ {E_0, ..., E_n}, e = ⟨e^0, ..., e^n⟩):
  V = {ℓ | ∃0 ≤ i ≤ n . e^i →_{E_i} e^{i'}}
  D = ∅
  for (i, ℓ, ℓ') such that 0 ≤ i ≤ n ∧ {ℓ, ℓ'} ⊆ V ∧ ℓ' ∈ strides[e^i][ℓ]:
    D = D ∪ (ℓ, ℓ')
  return (V, D) // (i.e., RA^e_E)
```

Listing 5.6. RA building procedure.

The procedure considers every pair of ready events in every automaton. In worst case, all events in A_E are ready in all automata $E_0, \ldots, E_n$ of E. In such a case, building the RA has a complexity of $O(n|A_E|^2)$. However, this worst case scenario is expected to be unlikely, since it would imply that all the events are always enabled.

In Listing 5.7, we show an algorithm for extracting the heuristic estimates for enabled events given a RA and a set of (already visited) marked states. The algorithm starts by computing the estimated distances from marked states to the ready events leading to them. Then, it continues performing a fix-point computation that propagates estimated distances to the events reachable by following the causal relations captured by the RA. The propagation of estimated values is repeated until it reaches an iteration in which no *fresh* value is found. In this way, we avoid analyzing all possible marked paths. Additionally, the algorithm keeps track of the *shortest* marked paths found for each automata E_i, which are then used to set the estimates for events $\ell \notin A_{E_i}$ that do not start a marked path in E_i.

For presentation purposes we assume the existence of the following elements:

- *strides*, a precomputed table indicating for every state which events lead to other states and events, plus in how many steps.
- `initTable`, a procedure that given a set of elements and a value, creates a table mapping each element in the set to the given value.
- `rank`, a function that produces a ranking with uncontrollable events in descending order followed by controllable events in ascending order. Events are compared by their heuristic estimates, that is, performing the lexicographical comparison over the estimates arranged in decreasing order.

The procedure is a fixed-point algorithm that iteratively propagates the sum of lengths in the paths reaching a marked state. At each iteration, at least one event is updated to its definitive value, since it considers the distance of the shortest path to a marked state. In worst case, a single event obtains its definitive value in every iteration, and this propagates to all the remaining events. Thus, in such a case the algorithm takes a quadratic number of iterations before converging to the fixed-point in addition to the initialization cost

```
procedure evaluateRA(RA^e_E = (V, D), M):
  initial = [⟨1,∞⟩_0, ..., ⟨1,∞⟩_n]
  estimates = initTable(V, initial)
  enabled = initTable({ℓ | ∀0 ≤ i ≤ n . e^i -ℓ->_{E_i} e^{i'}}, initial)
  shortest = initTable({i | 0 ≤ i ≤ n}, ⟨1,∞⟩)
  fresh = ∅
  for (e^i, ℓ, e^{i'}, d_i) ∈ strides such that 0 ≤ i ≤ n ∧ ℓ ∈ V ∧ e^{i'} ∈ M_{E_i}:
    m_i = if e^{i'} ∈ M: 0 else: 1
    estimates[ℓ][i] = min{estimates[ℓ][i], ⟨m_i, d_i⟩}
    if ℓ ∉ fresh:
      enqueue(fresh, ℓ)
  while fresh ≠ ∅:
    ℓ' = dequeue(fresh)
    for (i, ℓ, ℓ') such that 0 ≤ i ≤ n ∧ (ℓ, ℓ') ∈ D:
      ⟨m'_i, d'_i⟩ = estimates[ℓ'][i]
      ⟨m_i, d_i⟩ = ⟨m'_i, d'_i + max_j {strides[e_j][ℓ][ℓ']}⟩
      if estimates[ℓ][i] > ⟨m_i, d_i⟩:
        estimates[ℓ][i] = ⟨m_i, d_i⟩
        if ℓ ∉ fresh:
          enqueue(fresh, ℓ)
      if ℓ ∈ enabled:
        shortest[i] = min{shortest[i], ⟨m_i, d_i⟩}
  for (i, ℓ) such that 0 ≤ i ≤ n ∧ ℓ ∈ V \ A_{E_i}:
    if estimates[ℓ][i] = ⟨1,∞⟩:
      estimates[ℓ][i] = shortest[i] + ⟨0, 1⟩
  for ℓ ∈ enabled:
    enabled[ℓ] = estimates[ℓ]
  return rank(enabled)
```

Listing 5.7. RA heuristic evaluation.

of considering the runs leading to marked states. That is, the complexity of the procedure is $O((\sum_{i=0}^{n} |A_{E_i}||M_{E_i}|) + |A_E|^2)$.

5.4 Early Error Detection Optimization

In this section, we discuss a simple optimization to DCS based on early detection of errors. This optimization requires heuristics to report events known for certain to prevent reaching the goal. Such heuristics may underestimate the negative effects of an event, but may never overestimate such effects (i.e., never report false positives).

The optimization requires only a minimal modification to the DCS algorithm. That is, when consuming a recommendation, DCS has to check whether the event is flagged as an (early identified) error and invoke the error propagation

procedure for every such event. This may help to avoid futile exploration, and hence reduce the time required to obtain a solution.

Interestingly, both abstractions presented in the previous sections (i.e., the MA and the RA) fulfill this criteria. In particular, when the abstractions find the shortest distance to a marked state in an automaton to be infinite, we can safely conclude that the event prevents reaching the goal. Thus, this optimization comes at no additional cost.

The following result enunciates the error detection property for the MA.

Property 3 (MA Error Detection)**.** Let MA^e_E be the MA of the set of automata $E = \{E_0, \ldots, E_n\}$ from the state $e = \langle e^0, \ldots, e^n \rangle$ of E; and let ℓ an event enabled from e such that $MA^e_E(\ell) = \langle 1, \infty \rangle$. Then, ℓ prevents reaching a marked state of E.

Proof. The proof follows from Definition 12. Note that for $MA^e_E(\ell) = \langle 1, \infty \rangle$, it has to happen that $\langle 1, \infty \rangle$ is the maximum estimated distance for a given automaton E_i of E. That is, it has to be the case that there is no path starting with ℓ from a state e^i of E_i and reaching a state in M_{E_i}. Since there is no such path in the abstraction, there cannot be a run $e \xrightarrow{\ell \ldots \ell'}_E e_k$ with $e_k \in M_E$. Thus, ℓ prevents reaching a marked state of E_i, and consequently a marked state of E. □

The following result enunciates the error detection property for the RA.

Property 4 (RA Error Detection)**.** Let RA^e_E be the RA of the set of automata $E = \{E_0, \ldots, E_n\}$ from the state $e = \langle e^0, \ldots, e^n \rangle$ of E; and let ℓ_0 an event enabled from e such that $RA^e_E(\ell)$ contains a $\langle 1, \infty \rangle$ value at position $0 \leq i \leq n$. Then, ℓ prevents reaching a marked state of E.

Proof. The proof follows from Definition 17. In order to get a $\langle 1, \infty \rangle$ estimate for automaton E_i, it has to happen that $\ell \in A_{E_i}$ and that there is no path starting with ℓ and reaching a state of M_{E_i}. That is, after the occurrence of ℓ from e marked states of E_i become unreachable. Since ℓ prevents reaching a marked state of E_i, it also prevents reaching a marked state of E. □

Observe, that the error detection property for the RA only checks if a marked

state becomes unreachable in an individual automaton (i.e., not considering the interactions due to synchronization with other components). On the contrary, the MA considers (to some extent) the broader case when a marked state is unreachable because a required intermediate synchronization step cannot occur. However, in both cases the error identification is very conservative, and hence heuristics may fail to detect an error ahead of time. In such a case, the heuristics will report an overly optimistic underestimate of the distance to the marked state. Therefore, many errors will not be pruned by this approach, potentially requiring the exploration of the complete state space (i.e., worst case analysis is unaffected by the optimization).

5.5 Evaluation

In this section, we report on an evaluation for DCS. For the evaluation, we use the benchmark introduced in Chapter 3. We compare the results obtained by DCS with the results presented in Section 4.4, that is, taking advantage of the translations developed in Chapter 4. All the experiments were run on a desktop computer with an Intel i7-3770, with 8GB of RAM, and a time-out of 30 minutes.

5.5.1 Benchmark Comparison

In Figure 5.7 we show detailed results for each abstraction and case study. Each plot shows the combination of parameters n (horizontal axis) and k (vertical axis). Black represents a time-out (no out of memory or other kind of unsuccessful termination occurred during the execution of this evaluation).

In Figure 5.8.a we summarize the results reporting on the total solved cases. For ease of comparison we duplicate the results in Figure 4.2.a. In Figure 5.8.b we report on the total execution time in minutes (sum of times of every test up to – and including – time-outs), again including the results shown in Figure4.2.b.

From the results we can extract the following:

- The performance of DCS is (as expected) highly dependent on the

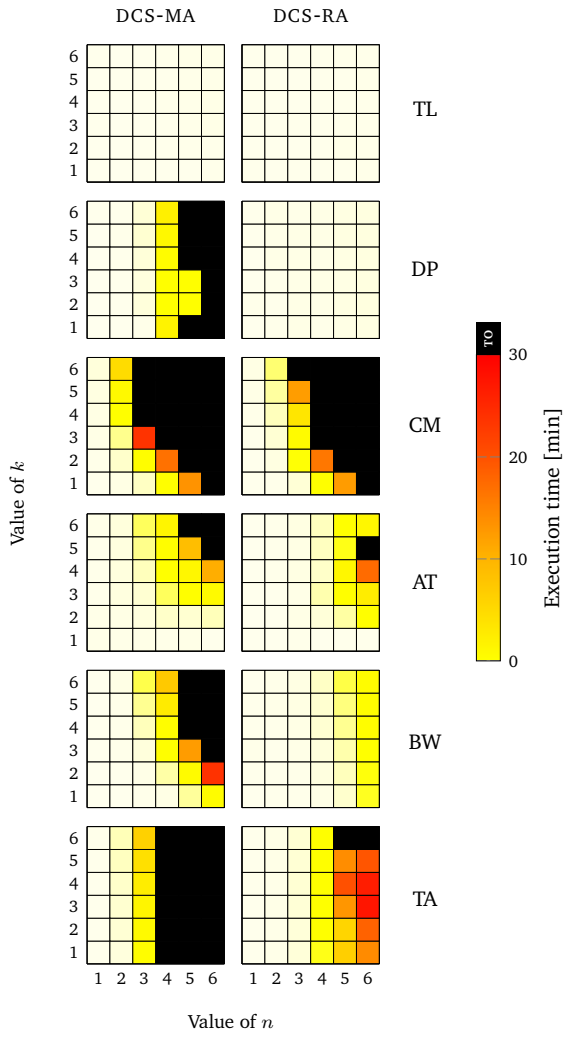
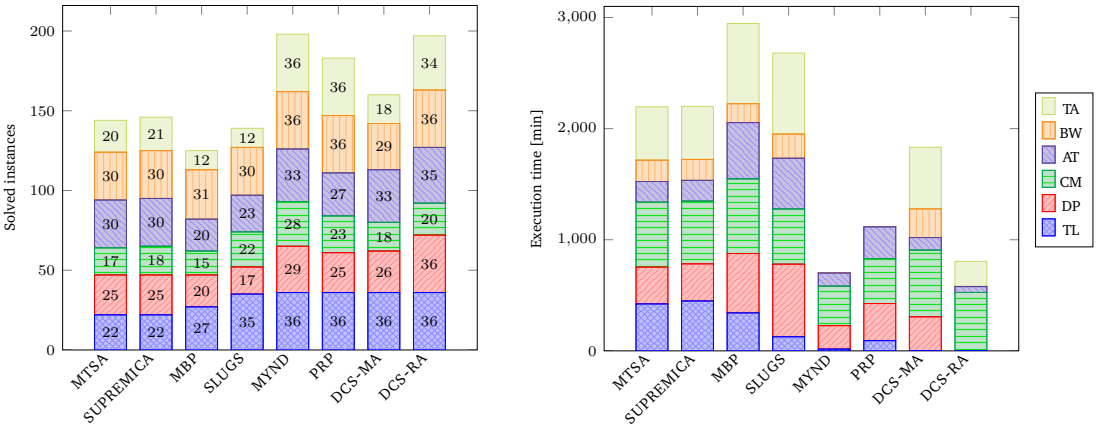


Figure 5.7. DCS detailed benchmark results.

heuristic. In particular, the RA performs consistently better than the MA for the selected benchmark.

- DCS performs comparably better than traditional supervisory control and reactive synthesis tools for the selected benchmark.
- The performance of DCS relying on the RA is comparable to state-of-the-art planning tools working under the encoding presented in Chapter 4.
- The different tools find challenges in different problems. For instance MYND takes more time to solve the DP problem than the TA problem, whereas in the case of DCS is the other way around.



(a) Total solved instances with DCS. **(b)** Total execution time with DCS.

Figure 5.8. Summarized benchmark results with DCS.

- The exponential complexity presented by the CM problem has a big impact on DCS. DCS solves the largest number of instances for each case except for CM and TA. Remarkably, MYND manages to solve many instances of these difficult problems.
- DCS faces worst case scenarios when problems do not have a solution (e.g., AT-6-5), because this forces an exhaustive exploration of the state space. That is, the conservative early error detection does not prune the exploration structure aggressively enough.

5.6 Summary and Results

In this chapter we present the DCS method, which looks for a supervisor by exploring the state space on-the-fly guided by domain-independent heuristics. We propose two heuristics which exploit the compositional nature of supervisory control problem specifications, providing informative estimates while avoiding the state explosion problem. Comparing our implementation with state of the art tools of three different disciplines tackling the synthesis problem, we found that DCS is among the top performers for the selected benchmark.

The risk of DCS is that the heuristics it uses could fail to properly guide the search, triggering an unnecessary large exploration of the state space. This

risk is shared with other techniques relying on informed search procedures, such as planning. Moreover, techniques based on symbolic representations with BDDs also rely on (very different) heuristics to obtain an ordering of variables, which contribute to a compact BDD. Some compositional approaches also use heuristics to decide the order in which to combine independently obtained results for different components. That is, approaches relying on heuristics share similar risks, which may hamper their ability to cope with the state blowup. All in all, computational hard problems such as the synthesis problem are not tractable by the application of brute force and, despite only being approximated, heuristics are empirically proven to be a powerful tool.

In its most general form, supervisory control looks for supervisors in potentially non-deterministic environments. As presented in this chapter, DCS only provides solutions for the deterministic case. We argue that synthesizing supervisors for deterministic environments is sufficient in many application domains. Hence, we are encouraged to work with the more confined set of features acceptable by the problem at hand, and leveraging efficient procedures taking this restrictions into account. Still, in the following chapter we show how to extend DCS to tackle non-deterministic problems which, of course, comes at a cost in complexity.

5.7 Resumen del capítulo 5

En este capítulo presentamos la Síntesis Dirigida de Controladores (DCS por sus siglas en inglés) para sistemas de eventos discretos. El método DCS explora el espacio de soluciones “sobre la marcha” guiado por una heurística independiente del dominio. Dada una función heurística que estima la distancia entre un estado y un objetivo (i.e., un estado marcado alcanzable infinitamente), DCS busca un supervisor computando la composición paralela sobre la marcha guiado por la heurística. En peor caso, la heurística guía la exploración en la dirección equivocada y se construye la composición paralela completa. Sin embargo, si la heurística guía correctamente la exploración puede hallarse una solución considerando sólo una porción reducida del espacio de estados.

Una ventaja del algoritmo es su simplicidad, dado que simplemente computa

un fragmento de la composición paralela paso a paso desde el estado inicial. Cuando un ciclo o un deadlock es descubierto, la información es propagada hacia atrás a través de la estructura de exploración marcando los estados en el camino. Cuando la propagación alcanza el estado inicial podemos determinar si un supervisor existe o no.

Una desventaja del algoritmo es que en peor caso puede computar la composición paralela completa (i.e., expandiendo todos los estados siguiendo todos los eventos habilitados). Con el costo adicional de computar la heurística para cada estado expandido, la complejidad del algoritmo es peor que su contraparte monolítica. Es decir, DCS podría tomar una cantidad exponencial de tiempo (i.e., EXPTIME). Por lo que la calidad de la información extraída para guiar la búsqueda resulta fundamental para el apropiado funcionamiento de la técnica.

Presentamos la abstracción monotónica (MA) y la heurística que induce. La MA fue previamente propuesta en el campo de model-checking dirigido como una adaptación a una heurística clásica utilizada en planificación. A pesar de las similitudes entre model-checking dirigido y DCS, la aplicación de la MA a control supervisor requiere de consideraciones adicionales, en particular lidiar con los requerimientos de controloabilidad y no-bloqueo.

El objetivo de la heurística es estimar la distancia de un estado al objetivo sin computar la composición paralela. Sin embargo, para poder proveer un estimado informativo los efectos de la sincronización deben tenerse en cuenta hasta algún punto. Para lograrlo la MA se comporta como si, luego de una transición, no sólo un nuevo estado es cubierto sino que el estado original no es abandonado. Es decir, la abstracción nunca descarta un estado una vez que alcanza un punto donde el estado es cubierto. Esto hace que la abstracción considere una conjunto de estados monotónicamente creciente, y de ahí obtiene su nombre.

A pesar de lucir muy diferente hay una relación fuerte entre un modelo y su abstracción. En particular, por cada ejecución en el autómata compuesto comenzando en el estado inicial, hay un camino en la abstracción. No obstante, lo contrario no es necesariamente cierto.

La función heurística estima la distancia al objetivo, considerando caminos

que comienzan en el estado inicial y llegan a un estado marcado. Sin embargo, para poder proveer una buena aproximación, la heurística también necesita tener en cuenta el momento en el que cada estado y evento en el camino es incluido en la MA, lo que llamamos la "generación" del estado (o del evento).

La generación es útil ya que permite detectar si un evento requiere pasos de sincronización intermedios antes de ser habilitado desde un estado dado. Es decir, si hay una diferencia entre las generaciones de un estado y un evento, podemos deducir la cantidad de pasos intermedios requeridos antes de poder realizar un paso. Esto permite a la heurística dar una mejor aproximación de la distancia al objetivos, de al que podría obtener simplemente tomando la longitud del camino.

Presentamos, también, la abstracción de eventos listos (RA) y la función heurística que induce. La RA representa una versión reducida del espacio de estados que descarta suficiente información como para evitar la explosión de estados, pero al mismo tiempo preserva suficientes detalles para permitir extraer estimados heurísticos útiles.

La RA sólo considera los eventos que están listos en los estados individuales de un estado compuesto, es decir, los eventos disponibles desde el estado actual de cada componente. Nos concentramos en los eventos listos porque nos permiten hacer una estimación local de la distancia a un estado marcado en cada automata y recursivamente qué otros eventos listos son necesarios para mover la exploración en esa dirección. Es decir, al computar la RA construimos un grafo de dependencias que indica qué eventos listos mueven la exploración hacia el objetivo y también en qué orden deben ocurrir.

La RA forma un grafo de tamaño polinomial con respecto a la cantidad de eventos en descripción composicional. El peor caso se evidencia cuando todos los eventos están listos en todos los automatas simultáneamente, pero es esperable que este escenario sea poco probable en la práctica porque implicaría que todos los eventos están siempre habilitados. Por lo tanto la RA puede construirse muy eficientemente.

Para proveer una distancia estimada al objetivo, consideramos todos los caminos en la RA que comienzan con un evento habilitado (i.e., listos en

todos los componentes intervinientes) y alcanzan un estado marcado. Tales caminos indican que es posible alcanzar un estado marcado siguiendo los eventos en el camino en orden, empezando con el evento actualmente habilitado. Luego, la distancia al objetivo puede ser aproximada por el número de pasos en cada ejecución de cada camino.

A diferencia de la MA, los estimados heurísticos extraídos de la RA son tuplas en lugar de valores simples (i.e., una distancia por cada automata). Esto se debe a que la RA no tiene en cuenta la cantidad de pasos de sincronización intermedios requeridos para seguir un camino. Por lo tanto, necesitamos criterios adicionales para realizar una comparación de grano fino. Mientras esto podría ser aplicado a la MA, no agregaría información dado que la MA ya considera (en alguna medida) los pasos intermedios requeridos.

Comparando nuestra implementación con herramientas en el estado del arte de las tres disciplinas que abordan el problema de la síntesis, encontramos que DCS se encuentra entre las herramientas con mejor desempeño para el benchmark seleccionado. El riesgo de DCS es que las heurísticas que usa pueden fallar en guiar la búsqueda, disparando una exploración del espacio de estados innecesariamente grande. Este riesgo es compartido con otras técnicas basadas en procedimientos de búsqueda informados como los de planificación. Incluso las técnicas basadas en representaciones simbólicas, frecuentemente utilizadas en síntesis reactiva, también se basan en heurísticas para obtener un ordenamiento de las variables, que contribuyen a mantener una compacta. También los acercamiento composicionales, tradicionales en control supervisor, usan heurísticas para decidir en qué orden combinar los resultados obtenidos independientemente por los distintos componentes. Es decir, todos los acercamientos basados en heurísticas comparten riesgos, que pueden obstaculizar su habilidad para lidiar con la explosión del espacio de estados.

En resumen, los problemas computacionalmente difíciles como el problema de la sítnesis no son tratables por la aplicación de fuerza bruta. A pesar de sólo ser aproximados, los procedimientos heurísticos representan una poderosa herramienta empíricamente comprobada.

6

Supervisory Control under Partially Observable and Non-Deterministic DES

6.1 Partially Observable and Non-deterministic DES

In this chapter, we extend the DCS technique to work on partially observable and non-deterministic DES. In this section, we start by extending the supervisory control formalization presented in Section 2.4.1 to the non-deterministic case. Interestingly, the behavior of a partially observable DES can be captured by means of non-determinism, and hence we also present a reduction from the former into the latter.

Non-deterministic supervisory control problems are described through non-deterministic automata. As in the deterministic case, the language accepted by these automata establishes the requirements for the supervisor to be.

Definition 18 (Non-deterministic Automaton)**.** A *non-deterministic automa-*

ton is a tuple $T = (S_T, A_T, \rightarrow_T, \bar{S}_T, M_T)$, where (similarly to the deterministic case, c.f. Definition 1):

- S_T is a finite set of states;
- A_T is the event set of T;
- $M_T \subseteq S_T$ is a set of marked states; while (unlike the deterministic case)
- $\rightarrow_T$ is a *non-deterministic* transition relation; and
- $\bar{S}_T \subseteq S_T$ is a non-empty set with the potential initial states.

Non-deterministic automata are a more general version of deterministic automata, where the differences lie within the transition relation and the initial states. In the non-deterministic case the transition relation $\rightarrow_T$ may provide multiple target states after a step. That is, a non-deterministic step may result in uncertainty on the supervisor's side regarding the exact state of the environment. For instance, non-determinism can be useful to model events that can silently fail, which may reach success or failure states. This uncertainty is also considered from the initial state $\bar{S}_T$, which is a set containing potential initial states.

Parallel composition (Definition 2) and compositional supervisory control problems (Definition 3) are naturally extended to the non-deterministic case by considering non-deterministic automata. A supervisor for such a problem would still have to guarantee the basic requirements, namely controllability and non-blockingness.

Partial observability brings the additional capacity of altering an automaton state through an internal event τ, invisible to the supervisor and other intervening components. Such internal events naturally appear where there is limited visibility of the environment, and can also arise form an abstraction of the system's behavior. Intuitively, the behavior of a partially observable system can be captured by means of non-determinism. Still, special care must be taken when reducing partial observability to non-determinism in order to preserve realizability.

When considering partial observability we introduce a special event τ with specific semantics with respect to communication.

Definition 19 (Parallel Composition Under Partial Observability)**.** The *parallel composition* ($\|$) of two non-deterministic automata T and Q is an associative symmetric operator that yields an automaton $T\|Q = (S_T \times S_Q, A_T \cup A_Q, \rightarrow_{T\|Q}, \bar{S}_T \times \bar{S}_T, M_T \times M_Q)$, where $\rightarrow_{T\|Q}$ is the smallest relation that satisfies the following rules:

$$\frac{t \xrightarrow{\ell}_T t'}{\langle t,q\rangle \xrightarrow{\ell}_{T\|Q} \langle t',q\rangle}\ell\in(A_T\setminus A_Q)\cup\{\tau\} \qquad \frac{q \xrightarrow{\ell}_Q q'}{\langle t,q\rangle \xrightarrow{\ell}_{T\|Q} \langle t,q'\rangle}\ell\in(A_Q\setminus A_T)\cup\{\tau\}$$

$$\frac{t \xrightarrow{\ell}_T t' \quad q \xrightarrow{\ell}_Q q'}{\langle t,q\rangle \xrightarrow{\ell}_{T\|Q} \langle t',q'\rangle}\ell\in(A_T\cap A_Q)\setminus\{\tau\}$$

In other words, the internal event τ does not cause an interaction in the parallel composition. Moreover, a supervisor for a partially observable environment T is a function $\sigma : (A_T\setminus\{\tau\})^* \mapsto 2^{A_T}$. That is, not only τ does not produce synchronization between components, but is also ignored by supervisors.

A direct approach to reduce partial observability to non-determinism is to apply τ-closure [36] over the automata. Such a reduction is known to preserve weak bisimilarity.

Notation 4 (Walk)**.** We denote by $t_0 \overset{\ell}{\Rightarrow}_T t_k$ that there exists a sequence of steps in automaton T, starting in state t_0 and reaching a state t_k after zero or more τ-transitions followed by exactly one ℓ transition (where ℓ could be τ) and again followed by zero or more τ-transitions; we call $t_0 \overset{\ell}{\Rightarrow}_T t_k$ a *walk*. More formally, there exists a sequence of steps such that:

$$t_0 \xrightarrow{\tau}_T \ldots \xrightarrow{\tau}_T t_i \xrightarrow{\ell}_T t_{i+1} \xrightarrow{\tau}_T \ldots \xrightarrow{\tau}_T t_k$$

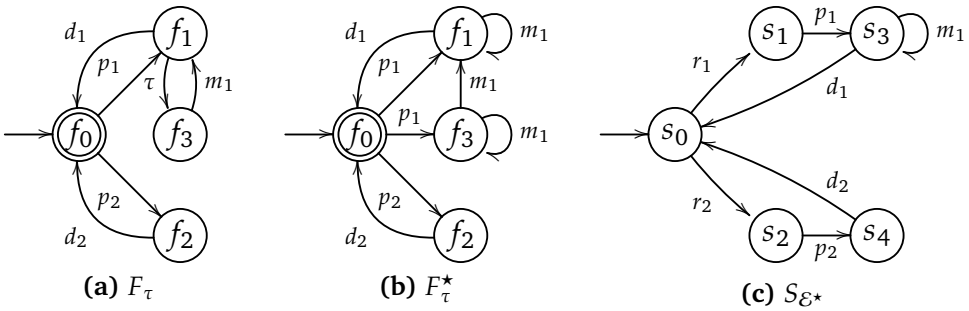
A *walk* resembles the notion of τ-closure as in [36], and it makes the definition of τ-removal for a non-deterministic automata simple.

Definition 20 (τ-removal)**.** Given a non-deterministic automaton $T = (S_T, A_T, \rightarrow_T, \bar{S}_T, M_T)$, we define its τ-*removed* version $T^\star = (S_T, A_T\setminus\{\tau\}, \rightarrow_{T^\star}, \bar{S}_{T^\star}, M_T)$, where:

- $\rightarrow_{T^\star} = \{t \overset{\ell}{\rightarrow}_{T^\star} t' \mid \ell \neq \tau \wedge t \overset{\ell}{\Rightarrow}_T t'\}$ is the τ-closure of the original transition relation $\rightarrow_T$; and
- $\bar{S}_{T^\star} = \bar{S}_T \cup \{t' \mid \exists \bar{t} \in \bar{S}_T \,.\, \bar{t} \overset{\tau}{\Rightarrow}_T t'\}$ is the set of initial states containing the original states plus those reachable through τ-transitions.

The definition of τ-removal produces a τ-free non-deterministic automaton by introducing a step between states if there is walk between them in the original automaton. The reduction works by removing τ-steps while capturing the same behavior by propagating non-determinism to previous non-τ-steps, and increasing the initial uncertainty for τ-steps originating in an initial state.

Example 8. *Recall the example from Figure 2.1 where two automata C and F model a manufacturing plant that works on-demand. Let us consider a variation of automaton F in which producing event p_1 may silently cause a failure. This failure can be fixed by performing maintenance, modeled with event m_1. However, the maintenance event does not guarantee fixing the issue, and hence it might be necessary to perform it numerous times. In Figure 6.1.a we depict automata F_τ exposing this behavior.*



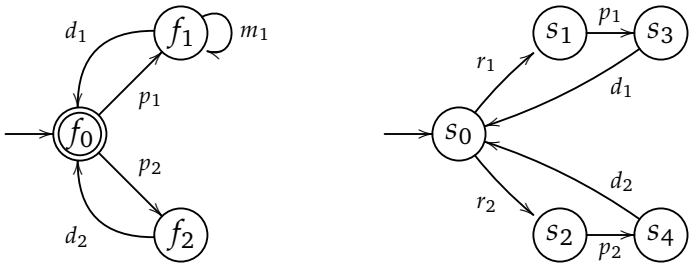
(a) F_τ **(b)** $F^\star_\tau$ **(c)** $S_{\mathcal{E}^\star}$

Figure 6.1. Non-deterministic Supervisory Control Problem Example.

Observe that from the initial state f_0 there is a deterministic step through event p_1 into state f_1. From state f_1 there is a τ-step into f_3 and a step through d_1 back to f_0. Thus, after the occurrence of p_1 a supervisor might not be able to discern if F_τ is at f_1 or f_3. Therefore, after p_1 a supervisor would need to provide a strategy guaranteeing that marked state f_0 is always reachable independently of the extact state it is in.

In Figure 6.1.b we depict the application of τ-removal over F_τ, namely $F_\tau^\star$. The behavior observed in $F_\tau^\star$ is equivalent to F_τ in the sense that after a potentially non-deterministic transition every possible continuation may ensue from one of the target states. In particular, F_τ and $F_\tau^\star$ are weakly bisimilar. Interestingly, the removal of τ-transitions causes p_1 and m_1 to exhibit a non-deterministic behavior.

Let $\mathcal{E}^\star = (\{C, F_\tau^\star\}, \{d_1, d_2, m_1, p_1, p_2\})$ be a compositional supervisory control problem. In Figure 6.1.c, we show a supervisor $S_{\mathcal{E}^\star}$ for $\mathcal{E}^\star$. Supervisor $S_{\mathcal{E}^\star}$ differs from the supervisor for the original problem $S_\mathcal{E}$, depicted in Figure 2.1.d, in that it allows event m_1 from s_3 in addition to d_1. That is, after p_1, the supervisor ignores whether it is at state f_1 or f_3, and hence enables d_1 leading to the marked state f_0 from f_1 and also enables m_1 leading to f_1 from f_3 (and potentially generating self-loops over f_1 and f_3). Note that m_1 does not guarantee effectively reaching f_0, yet this satisfies the non-blocking requirement. Remarkably, $S_{\mathcal{E}^\star}$ is also a solution in the partially observable plant $C \| F_\tau$.



(a) $F_\tau^\dagger$ **(b)** $S_{\mathcal{E}^\dagger}$

Figure 6.2. Invalid Trace Equivalent Reduction Example.

We could be tempted to produce a simpler automaton not necessarily bisimilar. For instance, a trace equivalent reduction $F_\tau^\dagger$ is shown in Figure 6.1.a (i.e., there is a run on a word w in F_τ if and only if there is also a run on w in $F_\tau^\dagger$). In this case there is a controllable choice between events p_1 and m_1 in state f_1. Thus, we may accept the supervisor $S_{\mathcal{E}^\dagger}$ depicted in Figure 6.1.b, which is valid for $C \| F_\tau^\dagger$ but not for $C \| F_\tau$ or $C \| F_\tau^\star$, since by not enabling m_1 it might lead us to a deadlock at state f_3. That is, considering $F_\tau^\dagger$ can make us reach the conclusion that a solution is valid when it is not (even when a solution

does not exist). For this reason, special care must be taken when reducing partial observability to non-determinism.

Observe that τ-removal exhibits the following useful properties, which point towards the fact that reducing a partially observable supervisory control problem into a non-deterministic τ-free version preserves realizability. We start by considering the property that a walk in an automaton T is substituted by a step in $T^\star$.

Lemma 1 (State Connectivity)**.** Given a non-deterministic automaton T, there is a walk on an event $\ell \in A_T \backslash \{\tau\}$ in T if and only if there is a corresponding step in $T^\star$. That is,

$$t \overset{\ell}{\Rightarrow}_T t' \Leftrightarrow t \overset{\ell}{\rightarrow}_{T^\star} t'$$

Proof. The proof is straightforward from the definition of τ-removal (Definition 20). □

Now we extend Lemma 1 to composed automata. This result states that we can apply τ-removal on the individual components, which will allow us to perform synthesis with a compositional approach.

Lemma 2 (Composition State Connectivity)**.** Given two non-deterministic automata T and Q, there is a walk in $T \| Q$ on an event $\ell \in A_{T\|Q} \backslash \{\tau\}$ if and only if there is a corresponding step in $T^\star \| Q^\star$. That is,

$$\langle t, q \rangle \overset{\ell}{\Rightarrow}_{T\|Q} \langle t', q' \rangle \Leftrightarrow \langle t, q \rangle \overset{\ell}{\rightarrow}_{T^\star\|Q^\star} \langle t', q' \rangle$$

Proof.

$\Rightarrow$ If $\langle t, q \rangle \overset{\ell}{\Rightarrow}_{T\|Q} \langle t', q' \rangle$ then one of the following holds:

- $\ell \in A_T \wedge \ell \notin A_Q$, thus $q = q'$ and by Lemma 1 $t \overset{\ell}{\rightarrow}_{T^\star} t'$, hence $\langle t, q \rangle \overset{\ell}{\rightarrow}_{T^\star\|Q^\star} \langle t', q' \rangle$;
- $\ell \notin A_T \wedge \ell \in A_Q$, thus $t = t'$ and by Lemma 1 $q \overset{\ell}{\rightarrow}_{Q^\star} q'$, hence $\langle t, q \rangle \overset{\ell}{\rightarrow}_{T^\star\|Q^\star} \langle t', q' \rangle$; or

- $\ell \in A_T \wedge \ell \in A_Q$, thus there must exist a sequence of τ-steps leading to a state $\langle t_i, q_j \rangle$ (i.e., T and Q progress independently through τ-steps) such that T and Q synchronize on ℓ, and which is followed by another sequence of τ-steps:

$$\langle t, q \rangle \xrightarrow{\tau}_{T\|Q} \ldots \xrightarrow{\tau}_{T\|Q} \langle t_i, q_j \rangle \xrightarrow{\ell}_{T\|Q} \langle t_i', q_j' \rangle \xrightarrow{\tau}_{T\|Q} \ldots \xrightarrow{\tau}_{T\|Q} \langle t', q' \rangle$$

  hence $\langle t, q \rangle \xrightarrow{\ell}_{T^\star\|Q^\star} \langle t', q' \rangle$.

$\Leftarrow$ If $\langle t, q \rangle \xrightarrow{\ell}_{T^\star\|Q^\star} \langle t', q' \rangle$ then one of the following holds:

- $\ell \in A_T \wedge \ell \notin A_Q$, thus $q = q'$ and by Lemma 1 $t \overset{\ell}{\Rightarrow}_T t'$, hence $\langle t, q \rangle \overset{\ell}{\Rightarrow}_{T\|Q} \langle t', q' \rangle$;
- $\ell \notin A_T \wedge \ell \in A_Q$, thus $t = t'$ and by Lemma 1 $q \overset{\ell}{\Rightarrow}_Q q'$, hence $\langle t, q \rangle \overset{\ell}{\Rightarrow}_{T\|Q} \langle t', q' \rangle$; or
- $\ell \in A_T \wedge \ell \in A_Q$, thus $T^\star$ and $Q^\star$ synchronize on event ℓ in state $\langle t, q \rangle$, that is $t \xrightarrow{\ell}_{T^\star} t'$ and $q \xrightarrow{\ell}_{Q^\star} q'$; then by Lemma 1 $t \overset{\ell}{\Rightarrow}_T t'$ and $q \overset{\ell}{\Rightarrow}_Q q'$; hence $\langle t, q \rangle \overset{\ell}{\Rightarrow}_{T\|Q} \langle t', q' \rangle$.

□

The following result states that a partially observable compositional supervisory control problem is realizable if and only if its τ-removed version is also realizable. Furthermore, a supervisor solution to one problem is also a solution to the other, thanks to the fact that supervisors ignore the internal event τ.

Theorem 5. Given a set of "partially observable" automata $E = \{E_0, \ldots, E_n\}$ (i.e., non-deterministic automata with the special event τ) and a partition of the event set A_C in controllable and uncontrollable events (i.e., $A_U = A_E \backslash A_C$), such that $\tau \notin A_C$. A supervisor σ is a solution for compositional supervisory control problem $\mathcal{E} = (E, A_C)$ if and only if σ is a valid supervisor for the corresponding τ-free control problem $\mathcal{E}^\star = (E^\star, A_C)$, with $E^\star = \{E_0^\star, \ldots, E_n^\star\}$.

Proof. We prove this by induction on the number of intervening components n. In the base case we consider a monolithic supervisory control problem,

and in the inductive case we consider the compositional supervisory control problem with n automata. Assuming the property holds for the composition of $n-1$ automata, it is enough to check that it holds for the composition of a single additional automaton. Analyses for base and inductive cases are analogous.

Let σ be a solution for $\mathcal{E}$, then for every word $w = \ell_0, \ldots, \ell_k$, such that $w \in \mathcal{L}^\sigma(E)$, there is a run $\bar{e} \overset{w}{\dashrightarrow}_E e'$ starting in some state $\bar{e} \in \bar{S}_E$. That is, there is a sequence of walks:

$$\bar{e} \overset{\ell_0}{\Rightarrow}_E \ldots \overset{\ell_k}{\Rightarrow}_E e'$$

By Definition 20 $\bar{e} \in \bar{S}_{E^\star}$, and by Lemma 2 each of these walks in E can be substituted with a step in $E^\star$:

$$\bar{e} \overset{\ell_0}{\rightarrow}_{E^\star} \ldots \overset{\ell_k}{\rightarrow}_{E^\star} e'$$

Thus, there is a run $\bar{e} \overset{w}{\dashrightarrow}_{E^\star} e'$, and hence $w \in \mathcal{L}^\sigma(E^\star)$.

Then, since σ is a solution for $\mathcal{E}$ it satisfies the requirements of Definition 3, and hence it is also the case that:

(1) $A_U \subseteq \sigma(w)$, that is σ does not disable uncontrollable events in $E^\star$, since it is a solution for $\mathcal{E}$; and

(2) there exists a word $w' \in (A_E \backslash \{\tau\})^*$ such that $e' \overset{w'}{\dashrightarrow}_{E^\star} e_m$ with $e_m \in M_{E^\star}$, that is w can be extended to reach a marked state in $E^\star$ in the same way as in E (i.e., substituting every walk in E with a step in $E^\star$).

The inverse direction is similar, but considering that states of $\bar{S}_{E^\star}$ are either in $\bar{S}_E$ or are reachable through τ-steps from some state of $\bar{S}_E$. Hence, a run on $E^\star$ is a sequence of steps, each of which can be substituted by a walk in E by Lemma 2.

Therefore, a solution for $\mathcal{E}$ is a solution for $\mathcal{E}^\star$ and vice versa. □

Summarizing, in this section we show how to reduce a partially observable compositional supervisory control problem into a non-deterministic one by applying a τ-closure procedure. The τ-closure of an automaton can be computed with Floyd's algorithm [76] in polynomial time (i.e., roughly cubic).

It has been shown that non-deterministic control problems can be reduced to the deterministic case by a determinization procedure [52], and hence leverage in existing synthesis techniques. However, determinization usually incurs in an exponential blowup of the state space. Furthermore, determinization cannot be applied compositionally, forcing the computation of the parallel composition ahead of time. Thus, we abstain from such approach. However, in the exceptional case that after τ-removal the obtained automata are deterministic (i.e., no non-deterministic transitions exist in the automata), we can rely on deterministic methods without additional cost.

Our τ-removal proposal can be applied to the individual components avoiding the computation of the parallel composition (i.e., delaying the potential exponential blowup to the exploration phase). In the following section we present an extension to DCS that takes as input a set of τ-free non-deterministic automata, which by performing an on-the-fly exploration might avoid the state explosion. Additionally, by relying on the above reduction our technique is also applicable in partially observable settings.

6.2 Directed Controller Synthesis under Non-determinism

In this section we present an extension to the DCS algorithm (introduced in Listing 5.1) for the non-deterministic setting (ND-DCS). The key difference is that instead of exploring the state space of the parallel composition, ND-DCS explores the space of beliefs. In this context a *belief* is a set of states in which the system could be at a certain point. That is, a belief b for a non-deterministic set of automata E is an element of 2^{S_E}.

For deterministic DES we are always certain about the current state. This is a special case in the non-deterministic setting in which beliefs are singleton sets. In general, in the presence of non-determinism we need to consider all the potential current states, and supervisors need to properly enforce control from any such state. Moreover, consecutive occurrences of non-deterministic events can further augment the level of uncertainty by progressively increasing the number of states in a belief. That is, for every state in a belief b, from which there is an enabled step through an event ℓ, we need to include its targets states in the following belief b'. Otherwise stated (with a slight abuse

of notation),

$$b \xrightarrow{\ell}_E b' \text{ where } b' = \{e' \mid \exists e \in b \,.\, e \xrightarrow{\ell}_E e'\}$$

Interestingly, events can also help reducing the uncertainty about the current state. Specifically, when we observe the occurrence of an event ℓ from a belief b, we know that ℓ could only have come from a state where it was enabled. Thus, if there are states in b, where ℓ is not enabled, we omit them in the following belief b'.

Recall that DCS explores the state space on-the-fly verifying that a marked state is always reachable. Likewise, ND-DCS will explore the belief space on-the-fly verifying that a "marked belief" is always reachable. We say that a belief is marked when all the states contained in it are marked. This is because there is no room for uncertainty if we are to guarantee that a marked state is always reachable.

In addition to the obvious replacement of states with beliefs (e.g., in *ES*, *recommendations*, etc.), we consider the following adaptations:

- The heuristic function returns a ranking for the events enabled in any of the states in the current belief b. This can be easily done by computing the heuristic function for every state in b, and taking the maximum estimate for events enabled in multiple states of b.
- The `expandNext` procedure expands all child states from every state in the current belief b consuming its first unused recommendation. That is, it produces a new belief b' "on-the-fly" by following a (potentially non-deterministic) transition from every state in b.
- The propagation procedures perform a backward search from a belief b back to its ancestors until the propagation is interrupted. The propagation of a goal from a belief b requires the confirmation that for every state $e \in b$ there is an event ℓ_e known to lead to a goal from e; and if ℓ_e is enabled in some other state $e' \in b$ then confirmation that ℓ_e leads to a goal from e' is also required. In turn, the propagation of an error or an undetermined status for an event ℓ, automatically degrades the status of an ℓ-step from any other state in b.

In worst case, the exploration traverses the complete belief space. Thus, ND-DCS is 2EXPTIME with respect to the size of the individual non-deterministic automata in a compositional control problem. That is, the non-deterministic setting is exponentially harder than the deterministic case. Furthermore, applying the heuristic function to every state in each explored belief also has a negative impact on the computational cost.

Example 9. *Let us consider $\mathcal{E}^\star = (\{C, F_\tau^\star\}, \{d_1, d_2, m_1, p_1, p_2\})$ the compositional supervisory control problem over automata $F_\tau^\star$ from Figure 6.1.b (representing a factory) and C from Figure 2.1.a (representing the factory's client). In Figure 6.3 we depict the complete exploration structure built by ND-DCS, indexing each belief with the number of the iteration in which it was expanded.*

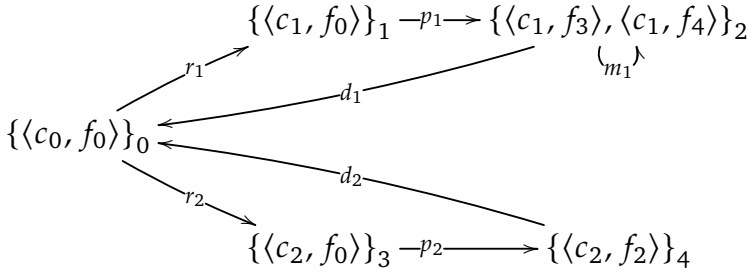


Figure 6.3. Exploration structure built by ND-DCS.

There is no uncertainty in the initial belief, thus ND-DCS starts from the singleton belief $\{\langle c_0, f_0\rangle\}$. From $\langle c_0, f_0\rangle$ the uncontrollable events r_1 and r_2 are available, and hence ND-DCS will explore them (i.e., by relying on either the MA or the RA we get a tie between r_1 and r_2 as shown in Figure 5.3.a and Figure 5.5.a respectively).

Let us suppose that ND-DCS explores r_1 first, expanding belief $\{\langle c_1, f_0\rangle\}$ at iteration 1. Since the transition through r_1 is deterministic, we again reach a singleton belief. From $\langle c_1, f_0\rangle$ controllable events p_1 and p_2 are enabled. Let us suppose that ND-DCS explores p_1 first (e.g., the heuristics derived from the MA and RA would effectively prioritize p_1 over p_2). Then, since the transition over p_1 is non-deterministic we expand belief $\{\langle c_1, f_3\rangle, \langle c_1, f_4\rangle\}$ at iteration 2, where there is uncertainty about the exact state of $F_\tau^\star$.

From $\langle c_1, f_4\rangle$ event m_1 is enabled and reaches $\{\langle c_1, f_3\rangle, \langle c_1, f_4\rangle\}$, since the transition over m_1 is also non-deterministic. The occurrence of m_1 closes a self-loop over the belief. From $\langle c_1, f_3\rangle$ event d_1 is enabled and reaches $\{\langle c_0, f_0\rangle\}$ closing a loop over a marked belief. Since uncontrollable event r_2 remains unexplored in the initial belief a CCC is not formed, and hence the belief is reopened for further exploration.

In the following, r_2 will be explored, expanding $\{\langle c_2, f_0\rangle\}$ at iteration 3. Then, p_2 will be explored, expanding $\{\langle c_2, f_2\rangle\}$ at iteration 4. And finally d_2 will be explored, closing a loop over $\{\langle c_0, f_0\rangle\}$. This time a marked CCC is formed and the propagation of goal flags intervening beliefs, including the initial belief. Therefore, ND-DCS *terminates and supervisor $S_{\mathcal{E}^\star}$ (as depicted in Figure 6.1.c) can be built by following the beliefs flagged as goals from the initial belief.*

As in the deterministic setting, this exhaustive exploration ensures the approach is sound and complete. Simply put, if there is a supervisor σ capable of mitigating the uncertainty arising from non-deterministic transitions, the exhaustive exploration of the belief space will reveal it. It is worth observing that in order to flag a belief b as a goal, the procedure needs to: (1) enable an event for every state in b; (2) enable every enabled uncontrollable event from the states in b; and (3) check that b belongs to a marked CCC (i.e., a word reaching b can be extended to reach a marked belief). Therefore, the initial belief $\bar{S}$ will be regarded as a goal only when it belongs to a marked CCC. Then, by following the transitions in such a CCC, ND-DCS can build σ.

6.3 Interaction Models

In this section, we identify alternative interaction models for supervisors, which are particularly relevant in the presence of non-determinism. The interaction model distinction is important because, in some settings, supervisors may have different capabilities to reduce the uncertainty about the system state, which has an impact on realizability.

Based on Interface Automata (IA) [77] we consider the additional require-

ment that a supervisor must not, at any point in time, allow controllable events not enabled in the plant at that point.

Definition 21 (IA control)**.** Let $\mathcal{E} = (E, A_C)$ be a compositional supervisory control problem; $b_w = \{e' \mid \exists \bar{e} \in \bar{S}_E \ . \ \bar{e} \overset{w}{\dashrightarrow}_E e'\}$ the set of all the possible states after a run on a word $w \in \mathcal{L}(E)$. We say that a supervisor σ solution for $\mathcal{E}$ is also a solution in the IA setting if and only if for every word $w \in \mathcal{L}^\sigma(E)$ it happens that:

$$\sigma(w) \subseteq \bigcap_{e \in b_w} \{\ell \in A_C \mid e \xrightarrow{\ell}_E e'\}$$

With an additional requisite, the IA interaction model defines a stronger version of the standard supervisory control problem. Hence, if there is a solution in the IA setting there is also a solution in the standard setting, but the converse is not always true. Interestingly, the distinction between these interaction models is irrelevant for deterministic control problems because the supervisor always knows the plant's state, trivializing the satisfaction of this requisite. We believe that for this reason, the IA interaction model has never been studied in classical supervisory control theory.

The IA setting is appropriate for problem domains in which the supervisor has the responsibility to enact controllable events. In such a case attempting to trigger a non-enabled event may result in a failure. Remarkably, it is simple to adapt ND-DCS to account for the IA interaction model. Specifically, when considering recommendations for controllable events, ND-DCS needs to focus only on the events enabled in *all* the states in the current belief. That is, events not in the intersection can be safely ignored (i.e., treated as errors), since they would allow the supervisor to potentially trigger an invalid event.

Example 10. *Consider the example in Figure 6.1.b in the context of the IA interaction model. Specifically, consider the non-deterministic automaton $F^\star_\tau$ representing a factory that may face a failure while producing event p_1, and the problem of supervising the plant $C \| F^\star_\tau$ without ever allowing non-enabled controllable events (where C is the automaton representing the factory's customer depicted in Figure 2.1.a).*

After exploring event p_1, states f_1 and f_3 are reachable non-deterministically,

and hence we face uncertainty about the state actually reached. It is easy to see that in the IA setting there exists no supervisor for the compositional supervisory control problem $\mathcal{E} = (\{C \| F_\tau^\star\}, \{d_1, d_2, m_1, p_1, p_2\})$*, since allowing* d_1 *from state* f_3 *would violate the IA requirement, while not allowing* d_1 *from* f_1 *would violate the non-blocking requirement.*

The proposed adaptation to ND-DCS *for considering the IA interaction model proceeds by exploring beliefs as in the standard case, depicted in Figure 6.3, up to iteration 2. At this point* ND-DCS *focuses on the intersection of events enabled from states in the belief* $\{\langle c_1, f_3\rangle, \langle c_1, f_4\rangle\}$*, which is empty. Therefore, the belief would be treated as a deadlock and an error would be propagated backwards to its ancestors (i.e.,* $\{\langle c_1, f_0\rangle\}$*). From* $\{\langle c_1, f_0\rangle\}$*, event* p_2 *remains unexplored, and hence* p_2 *would be explored next, expanding* $\langle c_1, f_2\rangle$*, which is also a deadlock. Again, an error would be propagated, but this time* $\{\langle c_1, f_0\rangle\}$ *would not have any remaining unexplored controllable events. Thus, the propagation would reach the initial belief* $\{\langle c_0, f_0\rangle\}$*, at which point the procedure would terminate reporting an error.*

The impact of interaction models on the synthesis problem remains a vastly unexplored area that we believe can lead to relevant software engineering insights. In particular, the standard supervisory control interaction model is a weaker version of the IA interaction model that arises naturally while abstracting a handshake communication mechanism. This handshake allows a supervisor to "sense" available controllable events.

Service oriented architectures, such as the Business Process Execution Language (BPEL) [78] or web-service orchestrators [79], provide a number of abstraction layers which can support a communication model based on handshakes. On the contrary, in cyber-physical systems is not sensible, in general, to presume a handshake between environment and actuators. Neither in software systems such as those based in the use of APIs specified using typestates [80].

In planning, the IA interaction model is implicitly assumed, since the expected solution policy must effectively instruct an autonomous agent. This discrepancy only becomes relevant under partial observability, which is studied in

a branch of planning called Partially Observable Non-Deterministic Planning (POND) [81]. Similarly, in reactive synthesis the IA interaction model is also assumed. Other potential sources of interaction models could be found in coordination models such as session types [82] or contracts [83] and in software architecture models like connector types [84], for which synthesis procedures could be used to alleviate development duties.

It is known that synthesis under partial observability is computationally more challenging than the deterministic case. Specifically, POND planning and reactive synthesis with partial information are known to be 2EXPTIME. Since a different set of tools and benchmark problems need to be considered for the partially observable setting, we leave the extension of the translation based approach of Chapter 4 and its performance evaluation as a matter for future work.

6.4 Resumen del capítulo 6

En este capítulo extendemos la técnica DCS para trabajar sobre sistemas de eventos discretos parcialmente observables y no-determinísticos. Empezamos extendiendo la formalización de control supervisor al caso no-determinístico. Resulta interesante que el comportamiento de los sistemas parcialmente observables pueden capturarse por medio del no-determinismo, por lo que también presentamos una reducción del primero en el segundo.

La observabilidad parcial conlleva la capacidad adicional de alterar el estado de un autómata mediante un evento interno, invisible al supervisor y los otros componentes. Tales eventos surgen naturalmente cuando hay una visibilidad limitada del ambiente y también pueden surgir a partir de una abstracción del comportamiento del sistema. Intuitivamente el comportamiento de un sistema parcialmente observable puede ser capturado por no-determinismo. Sin embargo, debe tenerse especial cuidado al reducir observabilidad parcial a no-determinismo con el fin de preservar realizabilidad.

Mostramos como reducir problemas de control supervisor composicionales con observabilidad parcial a problemas no-determinísticos, mediante la aplicación de un procedimiento de clausura. El procedimiento de clausura de un

autómata puede ser computado con el algoritmo de Floyd [76] en tiempo polinomial (i.e., de orden aproximadamente cúbico).

A su vez los problemas de control no-determinísticos pueden reducirse al caso determinístico mediante un proceso de determinización [52], para así aprovechar las técnicas de síntesis existentes. No obstante, la determinización incurre en una explosión exponencial del espacio de estados. Más aún, la determiniszación no puede aplicarse composicionalmente, forzando la construcción de la composición paralela. Por este motivo nos abstenemos de los acercamientos basados en reducciones, salvo en el caso excepcional en el que luego del proceso de clausura el autómata resultante sea determinístico. Ya que en este caso podemos utilizar los métodos determinísticos sin costo adicional.

Nuestra propuesta para remover eventos no-observables puede ser aplicada a los componentes individuales, evitando así la construcción de la composición paralela (i.e., retrasando la potencial explosión exponencial a la fase de exploración). Además, presentamos una extensión a DCS que toma como entrada un conjunto de autómatas no-determinísticos, sobre el cual se puede hacer una exploración "sobre la marcha" que evite la explosión de estados. Llamamos a esta extensión ND-DCS. Aplicando la reducción anteriormente mencionada, este método resulta aplicable, también, al caso parcialmente observable.

La diferencia principal de ND-DCS es que en lugar de explorar el espacio de estados resultante de la composición paralela, este procedimiento explora el espacio de creencias. En este contexto una *creencia* es un conjunto de estados en donde podría estar el sistema en un determinado punto del tiempo.

Recordemos que DCS explora el espacio de estados sobre la marcha, verificando en cada paso que un estado marcado sea siempre alcanzable. Del mismo modo ND-DCS explora el espacio de creencias sobre la marcha verificando que una "creencia marcada" sea siempre alcanzable. Decimos que una creencia es marcada cuando todos los estados contenidos en ella están a su vez marcados. Esto se debe a que no hay lugar a incertezas si debemos garantizar que el estado marcado es efectivamente alcanzable.

En peor caso la exploración atraviesa el estado de creencias completo. Por

lo que ND-DCS es 2EXPTIME con respecto al tamaño de los autómatas no-deterministícos individuales en el problema de control composicional. Es decir, el caso no-determinístico es exponencialmente más difícil que el caso determinístico. Además, aplicar la función heurística para cada estado en una creencia también conlleva un impacto negativo en el costo computacional.

Finalmente, identificamos modelos de interacción alternativos, que son particularmente relevantes en presencia de no-determinismo. La distinción de modelos de interacción es importante porque en distintos contextos los supervisores pueden tener diferentes capacidades de reducir la incerteza sobre el estado del sistema, lo que afecta directamente la realizabilidad.

Basados en los Autómatas de Interfaz (IA) [77] consideramos el requerimiento adicional de que un supervisor no pueda, en ningún momento, permitir un evento controlable no habilitado por la planta en ese momento. Con un requisito adicional, el modelo de interacción IA define una versión estrictamente más fuerte del problema de control supervisor estándar. Por lo que si hay una solución para el caso IA también hay una solución para el caso estándar, pero el inverso no es necesariamente verdadero. La distinción entre modelos de interacción es irrelevante para problemas determinísticos dado que en este caso los supervisores siempre conocen el estado de la planta, trivializando la satisfacción de este requisito. Creemos que por este motivo el modelo de interacción IA nunca fue estudiado dentro de la teoría clásica de control supervisor.

El modelo de interacción IA es apropiado en dominios donde el supervisor tiene la responsabilidad de actuar mediante los eventos controlables. En tal caso intentar ejecutar un evento no habilitado podría generar una falla. Resulta simple adaptar ND-DCS para considerar el modelo de interacción IA. Específicamente, cuando se consideran recomendaciones para eventos controlables ND-DCS necesita concentrarse solamente en los eventos habilitados en todos los estados de un estado de creencia. Es decir, los eventos que no se encuentran en la intersección pueden ser simplemente ignorados (i.e., tratados como errores), ya que permitirían que el supervisor ejecute un evento inválido.

El impacto de los modelos de interacción al problema de síntesis ha sido poco

explorado. Creemos que el estudio de estos modelos puede conducirnos a observaciones relevantes para la ingeniería del software. En particular el modelo de interacción estándar de control supervisor es una versión más débil que el modelo IA, que surge naturalmente cuando se abstrae un mecanismo de comunicación basado en handshakes. Este tipo de protocolos permiten al superviso "sensar" los eventos controlables disponibles.

Las arquitecturas orientas a servicios, como el Business Process Execution Language (BPEL) [78] o los orquestadores de web-services [79], proveen un número de capas de abstracción que pueden dar soporte a un modelo de comunicación basado en handshakes. Por el contrario, en los sistemas ciberfísicos no es razonable, en general, asumir un handshake entre el ambiente y los actuadores. Tampoco lo és en sistemas de software como los basados en el uso de APIs especificados mediante typestates [80].

7

Conclusions and Closing Remarks

7.1 Discussion and Recapitulation

The automatic synthesis of operational models from formal specifications promises to improve the quality of software engineering tasks by producing correct by construction solutions. This thesis studies the synthesis problem for DES from the supervisory control theory perspective. In this setting component-based systems are designed on the principle that each component contributes to achieve a sub-goal, and that the conjunction of these sub-goals achieves system requirements. That is, modular decomposition is central to capture the compositional nature of these systems, and is also a powerful tool for coping with system complexity at design time.

Reactive synthesis and automated planning are fields of study kindred to supervisory control. Hereinbefore, we review approaches from these areas and seek to shed light on their relation, in particular taking modular decompositions into account. Despite differences in representational aspects, we show how compositional supervisory control problems can be efficiently encoded

within these frameworks. Besides the theoretical contribution, these encodings effectively allow us to leverage on the advances made in these fields. We have empirically demonstrated that our translations allow us to solve compositional supervisory control problems with an efficacy that surpasses that of traditional supervisory control techniques.

The ulterior goal of this work is to promote cross-fertilization between these connected fields of study. In order to build in that direction we propose the DCS technique, which combines features of the top performing techniques. DCS builds upon modular specifications in order to extract (from polynomial-sized abstractions) guidance for informed search procedures. Our evaluation shows that DCS performs competitively with state-of-the-art techniques, although this is highly dependent on the heuristic induced by the abstraction used.

The synthesis problem brings along the state explosion problem, and hence in its most general form it is computationally intractable. However, practical applications can usually be tackled by state-of-the-art techniques, specially when relying on a constrained set of requirements. For this reason, we focus our study on the deterministic DES setting, since it is sufficient to accommodate many interesting application domains. Still, we also describe how our technique can be extended to the non-deterministic case, which of course incurs in higher computational costs.

Interestingly, the distinction between deterministic and non-deterministic DES also raises questions about the interaction models between plant and supervisor. We discuss that different domains may present incompatible interaction models, which need to be considered by the synthesis techniques since they affect realizability. The standard supervisory control interaction model is appropriate when it is reasonable to assume a communication model based on handshakes, which allows the supervisor to "sense" enabled events. We argue that some application domains may not exhibit this feature and that, in such a case, a stronger solution concept is required.

Summarizing, in this thesis we establish a link between compositional supervisory control theory, reactive synthesis and automated planning. Furthermore, we propose a synthesis method (DCS) inspired in the combination

of techniques from the different fields. Our evaluation places DCS between the top performers. Additionally, we discuss how to extend DCS to support non-deterministic and partial observable environments. We believe that the potential for cross-fertilization is still abundant and that, in spite of complexity barriers, advances in this direction will get us closer to handle practical problems from numerous application domains.

7.2 Future Work and Open Challenges

This thesis reveals a number of avenues for future work, regarding representational aspects, objectives, and algorithmic solutions among other topics. In this final section we explore some open ideas.

Naturally, translating supervisory control to reactive synthesis and planning raises the question of whether it is possible (and productive) to perform the inverse translation. Indeed, as mentioned in Section 2.1 there are a number of works relying in automata-based intermediate representations for reactive synthesis and planning. However, in general, these are monolithic representations, and hence subjected to a state blowup. In addition, the objectives of the different techniques are not easy to reconcile. For instance, reactive synthesis aims for general liveness goals (e.g., Büchi acceptance conditions), which cannot be directly encoded in the compositional supervisory control framework. Remarkably, in reactive synthesis attempts have been made to recover a modular decomposition by splitting conjunctive formulas [85]. However, such attempts are only able to solve safety-game goals, which are insufficient to express supervisory control objectives. In planning, modular finite sate representations have also been studied [86], yet it is unclear how to translate complex conditional effects and reachability goals into the supervisory control setting.

Likewise, our translations for the deterministic case raise the question of whether it is possible to perform similar translations in the non-deterministic/partially-observable setting. As mentioned in Section 6.3, the interaction model distinction becomes relevant when considering this setting. Moreover, the target frameworks would need to provide a way to "hide" the state from the supervisor. In POND planning [81], distinguished sens-

ing operations are added to disambiguate the observable state. That is, a non-deterministic supervisory control problem could be encoded to a POND planning problem by taking the interaction model and explicit observations into account. However, for reactive synthesis we know of no such extension. Additionally, it would be desirable to handle relabeling and hiding operators, which are commonly used in process calculi to introduce non-determinism. When used interleaved with parallel composition, these operators introduce a bigger challenge for translations since it becomes important to encode a non-flat (i.e., tree-like) composition structure. The translation would also need to allow the choice of multiple events per belief-state in order to meet the solution concept. In conclusion, such a translation would require non-trivial features, which are predictable to impact on efficiency and applicability.

On its own, the DCS method, opens paths for future research. An immediate question is whether new more effective heuristic functions can be devised. Such an heuristic could come from the combination of existing approaches, for instance by combining the RA and the MA (i.e., measuring local distance between events in the RA, but taking into account the information about required intermediate synchronization steps from the MA). However, we need to keep in sight that there is a delicate tradeoff between the computational cost of an heuristic and its precision. That is, we need to balance the cost of potentially futile exploration with respect to the cost of computing the heuristic. In this regard, we believe that cross-fertilization between the three fields may provide valuable insights to devise new informative heuristics.

The type of goals pursued by DCS is also a matter of consideration. Specialized optimizations for the stronger GR(1)-like goals or, alternatively, reachability objectives could be included. Soft goals, that is, optimization criteria associated with time or cost models [87], could also be studied. In particular, the directed nature of the exploration performed by DCS could prove to be an advantage for pursuing these kind of goals [88]. However, in supervisory control a cost model with a focus on concurrency and throughput would be required. In this context, we envision an any-time algorithm, that first finds a director, and then continues by doing an orderly exploration of the state space reopening relegated options in order to maximize throughput/concurrency. Despite the exponential blowup of the state space, such a technique

could still be applicable thanks to its any-time nature. That is, after a director is found, instead of returning the solution, we could continue improving the solution until a given time-out.

Furthermore, DCS could benefit from a more aggressive use of compositional analysis. For instance, when considering safety goals, states in individual components can be pruned until a safe region is obtained [85]. This reduction of states can in turn improve the performance of on-the-fly exploration and the quality of guidance provided by the heuristics. Moreover, since the performance of the technique relies heavily on the suitability of the heuristic, topological properties of the individual components could be analyzed in order to select the best fitting heuristic from a set of alternatives.

All things considered, we believe that there is a wide range of possibilities in which the combination of compositional analysis and informed search procedures can improve existing solutions for the synthesis problem. We hope that this work will foster further integration of results from the different disciplines, providing support for future software engineering challenges.

7.3 Resumen del capítulo 7

La síntesis automática de modelos operacionales a partir de especificaciones formales promete mejorar la calidad de las tareas de la ingeniería del software mediante la producción de soluciones correctas por construcción. Esta tesis estudia la síntesis para sistemas de eventos discretos desde la perspectiva de la teoría de control supervisor. En este contexto los sistemas son diseñados en base a componentes siguiendo el principio de que cada componente contribuye a lograr un sub-objetivo y que la conjunción de los sub-objetivos logra los requerimientos del sistema. Es decir, la descomposición modular es central para capturar la naturaleza composicional de estos sistemas, y además es una herramienta poderosa para lidiar con la complejidad de estos sistemas en tiempo de diseño.

La síntesis de sistemas reactivos y la planificación automática son campos de estudios cercanos al control supervisor. En este documento revisamos las técnicas usadas en estas áreas y buscamos comprender la relación entre las

disciplinas, en particular teniendo en cuenta las descomposiciones modulares. A pesar de las diferencias en aspectos de representación, mostramos como los problemas de control supervisor composicionales pueden ser eficientemente codificados dentro de los marcos de estos campos de estudio. Más allá de la contribución teórica, estas codificaciones permiten efectivamente aprovechar los avances hechos en estas áreas. Hemos demostrado empíricamente que nuestras traducciones nos permiten resolver problemas de control supervisor composicionales con una eficacia que supera a las técnicas de control tradicionales.

El objetivo ulterior de este trabajo es promover el intercambio entre estos campos de estudio relacionados. En este sentido proponemos la técnica DCS, que combina características de las técnicas con mejor desempeño. DCS se para sobre especificaciones modulares para extraer (de abstracciones de tamaño polinomial) guías para procedimientos de búsqueda informados. Nuestra evaluación muestra que DCS se desempeña competitivamente con respecto a las técnicas del estado del arte, pero resulta ser altamente dependiente de la abstracción utilizada y la función heurística inducida.

El problema de síntesis trae aparejado el problema de explosión del esapcio de estados, y por lo tanto es computacionalmente intratable en su forma más general. Sin embargo, aplicaciones prácticas pueden ser usualmente abarcadas por técnicas del estado del arte, especialmente cuando se restringen a un limitado conjunto de requerimientos. Por este motivo concentramos nuestro estudio en el caso de los sistemas de eventos determinísticos, ya que es suficientemente expresivo para acomodar muchos dominios de aplicación interesantes. A pesar de ello, también describimos como extender nuestra técnica al caso no-determinístico, que por su puesto incurre en un costo computacional mayor.

Resulta interesante que la distinción entre los casos determinísticos y no-determinísticos también genera preguntas sobre los modelos de interacción entre la planta y el supervisor. Discutimos que distintos dominios presentan modelos de interacción incompatibles, que necesitan ser considerados por las técnicas de síntesis ya que pueden afectar la realizabilidad. El modelo de interacción estándar es apropiado cuando es razonable asumir un modelo de comunicación basado en handshakes, lo que permite al supervisor “sensar”

los eventos disponibles. Argumentamos que algunos dominios de aplicación pueden no exhibir esta característica y que, en tales casos, un concepto de solución más fuerte es necesario.

En resumen, en esta tesis establecemos un vínculo entre la teoría de control supervisor composicional, la síntesis de sistemas reactivos y la planificación automática. Además, proponemos un método de síntesis (DCS) inspirado en la combinación de técnicas de las distintas áreas. Nuestra evaluación ubica a DCS entre las herramientas con mejor desempeño. Adicionalmente, discutimos como extender DCS para soportar entornos no-determinísticos y con observabilidad parcial. Creemos que, a pesar de las barreras de complejidad del problema, existe un gran potencial para el intercambio entre las disciplinas y que avances en esta dirección nos acercarán a resolver problemas prácticos para numerosos dominios de aplicación.

Appendix A

Finite State Processes

A.1 Modular Input Specifications

In this appendix we briefly describe the features of FPS [89] necessary to understand the case studies in Chapter 3. Technically, FSP is a process calculus designed to be easily machine readable. Importantly, FSP supports modular input specifications as the parallel composition of finite state processes.

In order to be analyzed, an FSP model has to be compiled first. We use MTSA [34] to compile and run the synthesis problems. If syntactical or grammatical errors are encountered the compiler output usually provides useful insights on how to fix them. If successful then the synthesis procedure will state whether a model satisfying the requirements exists. Additionally, MTSA supports exporting a compiled problem specification into different formats, including our translations to reactive synthesis and planning.

In the following we explain the FSP syntax and the MTSA constructs required for performing different analysis. We present only the subset of the features required by the benchmark presented in Chapter 3, for additional features consult MTSA's documentation.

A.2 Constants, ranges, sets and functions

Named constants, ranges, sets and functions can be defined as follows:

- `const ConstantId = Expression`
- `range RangeId = Expression..Expression`
- `set SetId = {element1,element2,...,elementN}`
- `def FunctionId(arg1,...,argN) = Expression`

Where identifiers must start with a capital letter, elements can be constants starting with lower case or other identifiers and expressions can be literals or simple arithmetic expressions. Functions are mere syntactic replacements thus recursion is not supported. From now on the initial capital letter rule for identifiers will be omitted for brevity.

Example 11.

```
// Number of philosophers.
const Philosophers = N

// Number of required etiquette steps.
const Steps = K

// Range of philosophers' Ids.
range Phil = 0..Philosophers-1

// Philosopher Id to left fork Id function.
def LeftP(p) = p

// Philosopher Id to right fork Id function.
def RightP(p) = (p+1) % Philosophers
```

A.3 Sequential Processes

A sequential process is defined by one or more local processes separated by commas. The definition is terminated by a full stop. Each process can include:

- Action prefix (->): `x -> P` describes a process that engages in an event `x` and then behaves as another process `P`.
- Choice (|): `x -> P | y -> Q` describes a process that can engage in either event, `x` or `y`.
- Re-labelling (/): `P/{x/y}` changes the name of the event `y` to `x` in `P`.
- Hiding (\): `P\{x}` renames event `x` as τ, a special event that is not shared between processes (normally events with the same name synchronize).
- Alphabet extension (+): `P+{x}` extends the alphabet of the process with a set of labels, which affects synchronization.
- Conditional (`if then else`): `if C then P else Q` describes a process that behaves as `P` if the condition `C` is true or as `Q` otherwise. Alternatively the keyword `when` can be used to specify a conditional behavior.
- Parameters: Optionally the processes can list parameters between parenthesis.
- Sequencing (`foreach`): `foreach [i:1..N] P(i)` represents the choice `(P(1)|...|P(N))`. Alternatively sequencing can be done in place during action prefixing: `(x[i:1..N] -> P(i))`.

Example 12.

```
// Monitor for a philosopher with a given Id.
Monitor(Id=0) = (eat[Id] -> Done),
  Done = (eat[Id] -> Done | eat.all -> Monitor).
```

Note: `STOP`, `ERROR` and `END` are primitive processes.

A.4 Parallel composition

The parallel composition 2 of one or more processes is a definition of a process preceded by `||`. The symbol `||` is also used for listing explicitly the processes being composed. A composed process can include:

- Prefixing (`:`): `x:P` prefixes every event of `P` with `x` (or each element of `x` if it is a range or set).
- Replication (`forall`): `forall [i:1..N] P(i)` is the parallel composition `(P(1)||...||P(N))`.

Example 13.

```
// DPPlant for the DP problem.
||DPPlant = (forall [p:Phil] (
  Philosopher(p) || Fork(p) || Monitor(p))).
```

A.5 Controller specification

The controller specification construct allows stating the different goals related to the model in the following form:

- `controllerSpec SpecId = {subGoals}`

Where `subGoals` can be any combination of the following:

- `marking = {SetId}`, which describes the property of reaching any of a given set of events (i.e., as by marking the states after such events).
- `controllable = {SetId}`, which indicates the set of controllable actions.
- `reachability`, which enables reachability analysis.
- `nonblocking`, which enables the non-blocking analysis.

Example 14.

```
// Control problem specification for DP
controllerSpec DPGoal = {
  controllable = {take[Phil][Phil]}
  marking = {eat.all}
  nonblocking
}
```

A.6 Analysis types

Special process can be defined in order to trigger different types of analysis:

- `controller ||Id = ProcessId~SpecId`, which triggers the monolithic synthesis procedure for a given specification.
- `heuristic ||Id = ProcessId~SpecId`, which triggers the directed synthesis procedure for a given specification.
- `minimal ||Id = ProcessId`, which triggers an analysis that returns a minimal automaton.

Example 15.

```
// Director for the DP problem.
heuristic ||DPDirector = DPPlant~{DPGoal}.
// Minimized director for the DP problem.
minimal ||MinDPDirector = DPDirector.
```

Note: the analysis type has to be selected either through the command line interface or using the graphical user interface.

Bibliography

[1] M. Young and M. Pezze. *Software Testing and Analysis: Process, Principles and Techniques* (John Wiley & Sons, 2005).

[2] M. Jackson. *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices* (ACM Press/Addison-Wesley Publishing Co., 1995).

[3] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems* (Springer-Verlag New York, Inc., 2006).

[4] P. J. Ramadge and W. M. Wonham. Supervisory Control of a Class of Discrete Event Processes. *SIAM Journal on Control and Optimization* **25**, 1987.

[5] A. Church. Applications of Recursive Arithmetic to the Problem of Circuit Synthesis. In *Summaries of the Summer Institute of Symbolic Logic*, 3–50 (1957).

[6] A. Pnueli and R. Rosner. On the Synthesis of a Reactive Module. In *Proc. of the 16th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, POPL, 179–190 (1989).

[7] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* **2**, 189–208, 1971.

[8] D. Nau, M. Ghallab and P. Traverso. *Automated Planning: Theory & Practice* (Morgan Kaufmann Publishers Inc., 2004).

[9] W. M. Wonham and P. J. Ramadge. Modular Supervisory Control of Discrete-Event Systems. *Mathematics of Control, Signals and Systems* **1**(1), 13–30, 1988.

[10] J. Huang and R. Kumar. Optimal Nonblocking Directed Control of Discrete Event Systems. *IEEE Tran. on Automatic Control* **53**, 1592–1603, 2008.

[11] R. Ehlers, S. Lafortune, S. Tripakis and M. Y. Vardi. Bridging the gap between supervisory control and reactive synthesis: Case of full observation and centralized control. In *Proc. of the 12th Int. Workshop on Discrete Event Systems*, WODES, 222–227 (2014).

[12] R. Bloem *et al.* Specify, Compile, Run: Hardware from PSL. *Electronic Notes in Theoretical Computer Science* **190**(4), 3–16, 2007.

[13] L. Ryzhyk and A. Walker. Developing a Practical Reactive Synthesis Tool: Experience and Lessons Learned. In *Proc. of the 5th Workshop on Synthesis*, SYNT@CAV, 84–99 (2016).

[14] B. Bonet and H. Geffner. Planning as Heuristic Search. *Artificial Intelligence* **129**, 2001.

[15] M. Ramírez, N. Yadav and S. Sardiña. Behavior Composition as Fully Observable Non-Deterministic Planning. In *Proceedings of the 23rd International Conference on Automated Planning and Scheduling*, ICAPS (2013).

[16] S. Mohajerani, R. Malik and M. Fabian. A Framework for Compositional Synthesis of Modular Nonblocking Supervisors. *IEEE Tran. on Automatic Control* **59**(1), 150–162, 2014.

[17] R. Bloem *et al.* RATSY – a New Requirements Analysis Tool with Synthesis. In *Proc. of the 22nd Int. Conf. on Computer Aided Verification*, CAV, 425–429 (2010).

[18] N. Lipovetzky and H. Geffner. Searching for Plans with Carefully Designed Probes. In *Proc. of Int. Conf. of Automated Planning and Scheduling*, ICAPS (2011).

[19] M. Y. Vardi and L. Stockmeyer. Improved Upper and Lower Bounds for Modal Logics of Programs. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC, 240–251 (1985).

[20] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli and Y. Saar. Synthesis of Reactive(1) Designs. *J. of Computer Systems Sci.* **78**(3), 911–938, 2012.

[21] J. Rintanen. Complexity of Planning with Partial Observability. In *Proc. 14th Int. Conf. on Automated Planning and Scheduling*, ICAPS (2004).

[22] O. Maler, A. Pnueli and J. Sifakis. On the Synthesis of Discrete Controllers for Timed Systems (An Extended Abstract). In *Symp. on Theor. Aspects of Computer Sci.*, STACS, 229–242 (1995).

[23] S. Cresswell and A. M. Coddington. Compilation of LTL Goal Formulas into PDDL. In *ECAI*, vol. 16, 985–986 (2004).

[24] F. Patrizi, N. Lipoveztky, G. D. Giacomo and H. Geffner. Computing Infinite Plans for LTL Goals Using a Classical Planner. In *Proc. of the 22nd Int. Joint Conf. on Artificial Intelligence*, vol. 3 of *IJCAI* (2011).

[25] A. Camacho, E. Triantafillou, C. J. Muise, J. A. Baier and S. A. McIlraith. Non-Deterministic Planning with Temporally Extended Goals: Completing the Story for Finite and Infinite LTL (Amended Version). In *Proc. of the Workshop on Knowledge-based Techniques for Problem Solving and Reasoning* (2016).

[26] A. Camacho, J. Baier, C. Muise and S. A. McIlraith. Bridging the Gap Between LTL Synthesis and Automated Planning. In *Workshop on Generalized Planning*, GenPlan (2017).

[27] S. Sardina and N. D'Ippolito. Towards Fully Observable Non-deterministic Planning as Assumption-based Reactive Synthesis. In *Proc. of the Int. Joint Conf. on Artificial Intelligence*, IJCAI, 3200–3206 (2015).

[28] S. Edelkamp, A. L. Lafuente and S. Leue. Directed explicit model checking with HSF-SPIN. In *Proc. of the 8th International SPIN Workshop on Model Checking of Software*, 57–79 (2001).

[29] C. Fritz, J. A. Baier and S. A. McIlraith. ConGolog, Sin Trans: Compiling ConGolog into Basic Action Theories for Planning and Beyond. In *Proc. of the 11th Int. Conf. on Principles of Knowledge Representation and Reasoning*, KR, 600–610 (2008).

[30] N. Yadav, P. Felli, G. D. Giacomo and S. Sardina. Supremal Realizability of Behaviors with Uncontrollable Exogenous Events. In *Proc. of the 23rd Int. Joint Conf. on Artificial Intelligence*, IJCAI, 1176–1182 (2013).

[31] R. Malik and H. Flordal. Yet Another Approach to Compositional Synthesis of Discrete Event Systems. In *9th Int. Workshop on Discrete Event Systems*, WODES, 16–21 (2008).

[32] R. C. Hill and D. M. Tilbury. Modular Supervisory Control of Discrete-Event Systems with Abstraction and Incremental Hierarchical Construction. In *Proc. of the 8th Int. Workshop on Discrete Event Systems*, WODES, 399–406 (2006).

[33] D. Ciolek, V. Braberman, N. D'Ippolito and S. Uchitel. Directed Controller Synthesis of discrete event systems: Taming composition with heuristics. In *Proc. of the 55th IEEE Conf. on Decision and Control*, CDC, 4764–4769 (2016).

[34] N. D'Ippolito, D. Fischbein, M. Chechik and S. Uchitel. MTSA: The Modal Transition System Analyser. In *Proc. of the 23rd IEEE/ACM Int. Conf. on Automated Software Engineering*, ASE (2008).

[35] S. Mohajerani, R. Malik, S. Ware and M. Fabian. Compositional synthesis of discrete event systems using synthesis abstraction. In *Control and Decision Conference*, CDC (2011).

[36] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM* **21**(8), 666–677, 1978.

[37] A. Gromyko, M. Pistore and P. Traverso. A Tool for Controller Synthesis via Symbolic Model Checking. In *Proceeding of the 8th Int. Workshop on Discrete Event Systems* (2006).

[38] A. Khalimov, S. Jacobs and R. Bloem. PARTY Parameterized Synthesis of Token Rings. In *Proc. of the 25th International Conference on Computer Aided Verification*, CAV, 928–933 (Springer Berlin Heidelberg, 2013).

[39] R. Ehlers and V. Raman. Slugs: Extensible GR(1) Synthesis. In S. Chaudhuri and A. Farzan (eds.) *Proc. of the 28th Int. Conf. on Computer Aided Verification*, CAV, 333–339 (2016).

[40] O. Kupferman and M. Y. Vardi. *Advances in Temporal Logic*, chap. Synthesis with Incomplete Informatio (Springer Netherlands, 2000).

[41] B. Bonet and H. Geffner. GPT: A Tool for Planning with Uncertainty and Partial Information. In *Proc. of IJCAI01 Workshop on Planning with Uncertainty and Incomplete Information*, 82–87 (2001).

[42] R. Mattmüller, M. Ortlieb, M. Helmert and P. Bercher. Pattern Database Heuristics for Fully Observable Nondeterministic Planning. In *Proc. of the 20th Int. Conf. on Automated Planning and Scheduling*, ICAPS, 105–112 (2010).

[43] C. Muise, S. A. McIlraith and V. Belle. Non-Deterministic Planning with Conditional Effects. In *Proc. of the Int. Conf. on Automated Planning and Scheduling*, ICAPS, 370–374 (2014).

[44] D. McDermott *et al.* PDDL—The planning domain definition language. Tech. Rep. DCS TR-1165, Yale Center for Computational Vision and Control, New Haven, Connecticut, 1998.

[45] J. McCarthy and S. A. I. Laboratory. *Situations, Actions, and Causal Laws*. Memo (Stanford Artificial Intelligence Project) (Comtex Scientific, 1963).

[46] R. Reiter. The frame problem in situation the calculus: A simple solution (sometimes) and a completeness result for goal regression. *Artificial Intelligence and Mathematical Theory of Computation*, 359–380 (Academic Press Professional, Inc., 1991).

[47] S. Jiang and R. Kumar. Supervisory Control of Discrete Event Systems with CTL* Temporal Logic Specifications. *SIAM J. Control and Optimization* **44**(6), 2079–2103, 2006.

[48] S. B. Akers. Binary Decision Diagrams. *IEEE Transactions in Computers* **27**, 1978.

[49] P. E. Hart, N. J. Nilsson and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *SIGART Bulletin* **37**, 28–29, 1972.

[50] S. Kupferschmid, J. Hoffmann, H. Dierks and G. Behrmann. Adapting an AI Planning Heuristic for Directed Model Checking. In *Proc. of the 13th Int. Conf. on Model Checking Software*, SPIN, 35–52 (2006).

[51] V. Alimguzhin, F. Mari, I. Melatti, I. Salvo and E. Tronci. On-the-Fly Control Software Synthesis. In *Proc. of the 20th Int. Symp. on Model Checking Software*, SPIN, 61–80 (2013).

[52] D. Ciolek, V. Braberman, N. D'Ippolito, S. Uchitel and N. Piterman. Interaction Models and Automated Control Under Partial Observable Environments. *IEEE Trans. Softw. Eng.* **43**(1), 19–33, 2017.

[53] E. Letier and W. Heaven. Requirements Modelling by Synthesis of Deontic Input-output Automata. In *Proc. of the 2013 Int. Conf. on Software Engineering*, ICSE '13 (IEEE Press, 2013).

[54] J. Klein, C. Baier and S. Klüppelholz. Compositional Construction of Most General Controllers. *Acta Inf.* **52**(4-5), 2015.

[55] A. Arnold, A. Vincent and I. Walukiewicz. Games for Synthesis of Controllers with Partial Observation. *Theoretical computer science* **303**(1), 2003.

[56] B. Bonet and H. Geffner. Planning with Incomplete Information as Heuristic Search in Belief Space. In *Proc. of the 6th International Conference on Artificial Intelligence in Planning Systems*, AIPS, 52–61 (AAAI Press, 2000).

[57] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proc. of the IEEE* **77**, 1989.

[58] J. Rintanen. Regression for Classical and Nondeterministic Planning. In *Proc. of the European Conf. in Artificial Intelligence*, ECAI, 568–572 (2008).

[59] J. Rintanen. Expressive Equivalence of Formalisms for Planning with Sensing. In *Proc. of the Int. Conf. on Automated Planning and Scheduling*, ICAPS, 185–194 (2003).

[60] C. Muise, S. A. McIlraith and J. C. Beck. Improved Non-deterministic Planning by Exploiting State Relevance. In *Proc. of the Int. Conf. on Automated Planning and Scheduling*, ICAPS, 172–180 (2012).

[61] A. Cimatti, M. Pistore, M. Roveri and P. Traverso. Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking. *Artificial Intelligence* **147**(1-2), 35–84, 2003.

[62] W. M. Wonham. Notes on Control of Discrete-Event Systems. Dep. of Electrical and Comp. Engineering, University of Toronto, 1999.

[63] D. Lehmann and M. O. Rabin. On the Advantages of Free Choice: A Symmetric and Fully Distributed Solution to the Dining Philosophers Problem. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, 133–138 (1981).

[64] R. K. L. Ko. A Computer Scientist's Introductory Guide to Business Process Management (BPM). *Crossroads* **15**(4), 4:11–4:18, Jun. 2009.

[65] F. T. Durso and C. A. Manning. Air traffic control. *Reviews of human factors and ergonomics* **4**(1), 195–244, 2008.

[66] B. Srivastava and J. Koehler. Web service composition-current solutions and open problems. In *ICAPS 2003 workshop on Planning for Web Services*, vol. 35, 28–35 (2003).

[67] J. G. Thistle. Supervisory Control of Discrete Event Systems. *Mathematical and Computer Modelling* **23**(11), 25–53, 1996.

[68] E. Filiot, N. Jin and J.-F. Raskin. Antichains and compositional algorithms for LTL synthesis. *Formal Methods in System Design* **39**(3), 261–296, 2011.

[69] J. G. Thistle and W. M. Wonham. Control of Infinite Behavior of Finite Automata. *SIAM J. of Control Optimization* **32**(4), 1075–1097, 1994.

[70] J. Huang and R. Kumar. Directed Control of Discrete Event Systems for Safety and Nonblocking. *IEEE Trans. Automation Science & Engineering* **5**, 2008.

[71] P. Gohari and W. M. Wonham. On the complexity of supervisory control design in the RW framework. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* **30**(5), 643–652, 2000.

[72] N. J. Nilsson. *Artificial Intelligence: A New Synthesis* (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998).

[73] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach* (Prentice Hall Press, Upper Saddle River, NJ, USA, 2009), 3rd edn.

[74] R. Tarjan. Depth-first search and linear graph algorithms. In *12th Annual Symposium on Switching and Automata Theory*, SWAT, 114–121 (1971).

[75] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numer. Math.* **1**(1), 269–271, 1959.

[76] R. W. Floyd. Algorithm 97: Shortest Path. *Communications of the ACM* **5**(6), 345–, 1962.

[77] L. de Alfaro and T. A. Henzinger. Interface Automata. In *Proc. of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, ESEC/FSE-9 (ACM, 2001).

[78] R. M. Dijkman, M. Dumas and C. Ouyang. Semantics and Analysis of Business Process Models in BPMN. *Information and Software Technology* **50**(12), 1281–1294, 2008.

[79] H. Foster, S. Uchitel, J. Magee and J. Kramer. Model-based Verification of Web Service Compositions. In *ASE* (2003).

[80] K. Bierhoff and J. Aldrich. Lightweight Object Specification with Typestates. In *Proc. of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, ESEC/FSE-13 (ACM, 2005).

[81] C. Weber and D. Bryce. Planning and Acting in Incomplete Domains. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling, ICAPS 2011, Freiburg, Germany June 11-16, 2011* (2011).

[82] K. Honda, V. T. Vasconcelos and M. Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Proc. of the 7th European Symp. on Programming: Programming Languages and Systems*, ESOP '98 (Springer-Verlag, 1998).

[83] G. Castagna, N. Gesbert and L. Padovani. A Theory of Contracts for Web Services. *ACM Trans. Program. Lang. Syst.* **31**(5), 2009.

[84] N. R. Mehta, N. Medvidovic and S. Phadke. Towards a Taxonomy of Software Connectors. In *Proc. of the 22nd Int. Conf. on Software Engineering*, ICSE '00 (ACM, 2000).

[85] A. Bohy, V. Bruyère, E. Filiot, N. Jin and J.-F. Raskin. Acacia+, a Tool for LTL Synthesis. In *Proc. of the 24th Int. Conf. on Computer Aided Verification*, CAV, 652–657 (2012).

[86] C. Bäckström. Equivalence and Tractability Results for SAS+ Planning. In *Proc. of the 3rd Int. Conf. in Principles of Knowledge Representation and Reasoning*, KR (1992).

[87] R. Kumar and V. K. Garg. Optimal Supervisory Control of Discrete Event Dynamical Systems. *SIAM Journal on Control and Optimization* **33**(2), 419–439, 1995.

[88] P. Haslum and H. Geffner. Admissible Heuristics for Optimal Planning. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems*, AIPS, 140–149 (2000).

[89] J. Magee and J. Kramer. *Concurrency: State Models &Amp; Java Programs* (John Wiley & Sons, Inc., 1999).