INFERENCE OF SAFE APPROXIMATE MODELS FOR SYSTEM COMPOSITION

Teză destinată obținerii
titlului științific de doctor inginer
la
Universitatea "Politehnica" din Timișoara
în domeniul CALCULATOARE ȘI TEHNOLOGIA INFORMAȚIEI
de către

Casandra Holotescu

Conducător științific:	Prof. univ. Dr. Ing. Horia Ciocârlie
Referenți științifici:	Prof. univ. Dr. Dana Petcu
	Prof. univ. Dr. Ing. Mircea Petrescu
	Prof. univ. Dr. Ing. Vladimir-Ioan Crețu

Ziua susținerii tezei: 26.04.2013

Seriile Teze de doctorat ale UPT sunt:

1. Automatică
2. Chimie
3. Energetică
4. Ingineria Chimică
5. Inginerie Civilă
6. Inginerie Electrică
7. Inginerie Electronică și Telecomunicații
8. Inginerie Industrială
9. Inginerie Mecanică
10. Știința Calculatoarelor
11. Știința și Ingineria Materialelor

Universitatea "Politehnica" din Timișoara a inițiat seriile de mai sus în scopul diseminării expertizei, cunoștințelor și rezultatelor cercetărilor întreprinse în cadrul școlii doctorale a universității. Seriile conțin, potrivit H.B.Ex.S Nr.14/14.07.2006, tezele de doctorat susținute în universitate începând cu 1 octombrie 2006.

România, 300159 Timișoara, Bd. Republicii 9,
tel. 0256 403823, fax. 0256 403221
e-mail: editura@edipol.upt.ro

Acknowledgements

I would like to thank my advisor, Prof. Horia Ciocârlie, for all the support he has shown me all these years, for all his good advice and encouragement, and for all his wisdom.

My gratitude goes to Prof. Dana Petcu, Prof. Mircea Petrescu and Prof. Vladimir-Ioan Crețu for promptly accepting to be the members of my Ph.D. examination commitee, and to Prof. Vasile-Stoicu Tivadar for accepting to chair the examination. As the head of Institute eAustria, Prof. Dana Petcu has also offered me material support during my Ph.D., and I thank her.

I am very grateful to Marius Minea, who has taken a lot of time to review my papers and my dissertation, and whose constructive criticism has made me a decent researcher and a better, braver, more logical and more rational (although not entirely) human being. Many thanks go to Prof. Ioan Jurca and Ioana Sora, who have also reviewed this dissertation and provided me with valuable feedback and advices.

I would also want to thank my colleague Mihai Balint, who has been the best "academic brother" one could hope for, and also a source of good cheer and valuable research experience. It probably wasn't easy working in the same office with the emotional storm that I am, but Mihai proved endless patience and empathy. Similar qualities have also shown Cristina Marinescu, Dan Cosma, Alexandru Gyori, Petru Mihancea and Petre Mierlutiu, and similar feelings of gratitude go towards them as well. It was so fun having you guys around, and thank you for all the teaching and research experience you've been kind enough to share with me. Gossip was highly appreciated as well.

Special thanks go to my whole family, who has shown me unconditional love and support through my darkest times. I would have probably been lost to my doubts and fears without their faith and warmth, so I'm truly happy that they have

somehow managed to stand by me all these strange, challenging years.

And, most of all, I would like to thank God for throwing me on such an exciting ride and for giving me enough strength and stamina to last through it.

This dissertation has been partially supported by the strategic grant POSDRU 6/1.5/S/13, (2008) of the Ministry of Labour, Family and Social Protection, Romania, co-financed by the European Social Fund – Investing in People, by the European FP7-ICT-2007-1 project 216471, AVANTSSAR: Automated Validation of Trust and Security of Service-oriented Architectures and by the European FP7-ICT-2009-5 project no. 257876 SPaCIoS: "Secure Provision and Consumption in the Internet of Services".

Holotescu, Casandra

Inference of Safe Approximate Models for System Composition

Teze de doctorat ale UPT, Seria 14, Nr.15, Editura Politehnica, 2013, 237 pagini, 176 figuri, 14 tabele.

Keywords: components, model learning, controllability, adaptation

Abstract,

The scope of this thesis is in the field of software engineering and formal methods. A core issue in component-based software engineering is how to rigorously build a system from off-the-shelf software components. This is done by automatic component adaptation and composition, which need behavioural models of all components. We have developed an approximate model inference technique for black box asynchronous software components. In contrast to existing techniques that assume only controllable behaviour, our method addresses the challenges raised by uncontrollable events, which cannot be enabled or disabled at will. The models inferred are safe with respect to black box controllability, i.e. they can be reliably used to control the real components.

Contents

Chapter 1

Introduction

1.1 Background and Motivation

Component based software engineering has emerged as a promising trend both in today's academic and industrial landscape. Its main benefit lies in reducing the actual length of the software development process: instead of designing, developing, testing and integrating several software modules, a desired system is composed out of reusable, off-the-shelf components. This makes for a faster and more flexible way of creating software systems.

However, integrating third party components can be a difficult and error prone task. If we want to build good software, we have to use well-tested, reliable components, and we have to make sure they interoperate correctly. In most cases, composing off the shelf components is far from straightforward and several mismatches appear at functional, behavioural, signature and quality of service levels [9]. In order to address this very frequent issue, specific adaptors have to be built, and, since building these adaptors by hand is usually a hard and tedious task, tools and techniques for automating system composition are needed.

In the area of behavioural adaptation, important advances have been made in the last two decades. Since the seminal work of Yellin and Strom [84], formal specifications of component behaviour have been used to obtain adaptors able to ensure correct interoperability at behavioural level. Such an adaptor is a specific component-in-the-middle that coordinates the interactions in the system for the purpose of achieving the desired functionality.

While early approaches, such as [72], were aiming for deadlock-free component interaction, later solutions such as [5] or [23] provide fully automated, tool-based adaptor generation, and the resulting adaptors also coordinate the system with the goal of satisfying desired temporal logic properties. Both desired properties and component behaviour are modelled by means of the same formalism, be it finite state machines, Büchi automata or labeled transition systems, and the composition is considered successful when the adapted system behaviour is simulated by the specified goal.

Behavioural system adaptation can also be easily reduced to a control problem [67], if we see the adaptor as a controller over the system plant, enabling transitions by message forwarding and disabling them, whenever necessary, by message consumption.

However, formally specified component behaviour is, unfortunately, still far from becoming a mainstream industrial practice. Therefore, situations when incompletely specified, black box components have to be correctly assembled to create a larger system are not at all uncommon. As none of the classic adaptor synthesis solutions can be applied, building a correct and reliable system under these circumstances is a challenging task.

So, how can the problem of correct software composition can be addressed under these circumstances? One solution is to first learn the models of the black box components by using an existing regular inference algorithm, such as L* [4]. However, learning a complete model for a component is hard, and implies a large amount of expensive queries applied to the black box component. Therefore, it is only feasible to infer approximate models of the black box components involved, under some assumptions, like knowing an upper bound for the model state size, etc. [44, 64]. This works fine for components that are completely controllable from outside, but what happens if we have to integrate components that also have uncontrollable events, i.e. events that cannot be enabled or disabled by an external agent?

Assume we have black box components that send and receive messages asynchronously. Also assume that for these components the message send events are uncontrollable, i.e. it is impossible to directly force the occurrence of a send event by some external action. If we want to integrate them into a system that would comply to a certain temporal property, we need to know everything about their uncontrollable behaviour, so that the system adaptor would be able to properly react to any uncontrollable event occurring in the system. However, since learning a complete model of the component can be infeasible, we have to

be able to build an overapproximation of the uncontrollable behaviour from any limited set of executions.

Therefore, the research questions asked are the following:

- How should learned models approximate the real behaviour of the black box component so that the adaptor to be generated for these models can enforce a safe behaviour on the real system ?
- How can we learn such safe approximate models for components whose event set contains both controllable and uncontrollable events?
- How can we control and explore the behaviour of such components in an efficient way, assuming the desired system would have to comply to a certain temporal property?

1.2 Main Contributions

1.2.1 Goal

The goal of our work is learning behavioural models for black box software components whose alphabet contains both controllable and uncontrollable events. The model inference is specifically done for compositional purposes. The learned models are to be used to compose a system under a specified safety property. Composition takes place by generating a controller for the system, that will use the controllable events in the models to disable any execution that violates the desired property. Therefore, the models obtained have to be reliable, i.e. to contain only observed controllable transitions, that can actually be enabled at runtime, and, also, to contain all the uncontrollable transitions in the real component's behaviour. Also, the learned models do not need to contain all the behaviour of the black box component, but only the subset of this behaviour that is relevant to the specified system property.

Thus, the goal and the main contribution of this thesis is developing an active, online model learning method specifically adapted to black box components that exhibit uncontrollable behaviour. The models learned are safe approximations of the real black box behaviour, i.e. overapproximations of its uncontrollable behaviour and underapproximations of its controllable behaviour, thus the system can be safely composed by generating a controller to enforce the desired specifi-

cation. For this purpose, we have developed two runtime behaviour exploration techniques and a model building method that have all been implemented in the BASYL (Black box ASYnchronous Learning) tool.

1.2.2 Centralized Behaviour Exploration

The centralized behaviour exploration method actively explores the global behaviour of the system. The runtime behaviour is observed at a global level, so all components in the desired system are executed together. However, the model of each black box component is inferred individually, using projections of the global execution traces on each component's alphabet of events.

Considering the desired temporal property, we delegate the task of coordinating the system to a proactive component-in-the-middle, a proactive adaptor. This adaptor monitors the components in the system and controls their interactions, by intercepting sent messages and forwarding or consuming them. Thus, the system is forced to only execute the behaviour needed to be learned, that is the behaviour relevant to the desired property. This runtime behaviour exploration process ends when either all relevant traces of bounded length have been explored, or when a certain cost limit has been reached.

1.2.3 Distributed Behaviour Exploration

The distributed behaviour exploration method relies on a localization of the behaviour exploration process in order to better fit the natural structure of distributed systems.

For each component in the system a local instance of a proactive adaptor is created, and the behaviour of each black box component is explored locally, together with the models of other components in the system. This allows for a parallelization of the model inference process, since in this case the components are executed and monitored independently from each other.

1.2.4 Cycle-Oriented Optimization

Both centralized and distributed behaviour exploration methods can be optimized to prioritize the runtime exploration of cyclic scenarios over any other

relevant behaviour. The aim of this prioritization is to achieve an early cycle identification in the learned models.

This is important, as many execution scenarios are either cyclical, or part of larger loops. The relevant cycles of the specification automaton, i.e. the cycles involved in specified usage scenarios, are projected on the alphabet of each black box component. Then, at runtime, the execution is specifically and repeatedly steered towards these loops. Identifying these cycles is necessary for the resulting composed system to allow the execution of cyclic scenarios.

1.2.5 Model Building

The model building phase takes place after the runtime behaviour exploration process ends, and it makes use of all observations gathered so far. These observations include both positive and negative sample traces, i.e. sequences of events determined as included, respectively not included in the behavioural language of the component under learning.

Most importantly, no matter when the behaviour exploration stops, the model building algorithm always results in a safe approximation of the real black box behaviour, that is the controllable behaviour of the component is always underapproximated by the built model, while the uncontrollable behaviour of the component is overapproximated. This allows for the approximate model to be reliably used for adaptor synthesis.

Precise models can also be obtained, when the black box behaviour is completely explored to a certain predefined depth.

1.2.6 Tool Support: `BASYL`

In order to empirically validate our approach, we have provided tool support for the theoretical solutions we have elaborated.

We have implemented our behaviour exploration and model building techniques in the `BASYL` tool, which is entirely written in Java. `BASYL` works at a high abstraction level, therefore it needs specific drivers for each technology used by real software components.

The learning techniques presented in this work have been tested, using BASYL, on a number of case studies from both model inference and component based software engineering literature. The results of the set of performed experiments have shown the validity of our thesis: BASYL can learn safe approximate models for components with uncontrollable behaviour.

Chapter 2

Preliminaries

2.1 Component Based Software Engineering

The concept of component was first introduced in 1968 by McIlroy [62], in a time when software reuse was limited to ad-hoc reuse of subroutines. By analogy with mechanical or hardware components, the vision of software components described in [62] was to have standard catalogs of mass produced, flexible and largely reusable routines from which the programmer to be able to choose the most suitable options for his project, given catalog descriptions based on various parameters (cost, performance, etc.).

Later on, this vision was enriched by the fast development of the object-oriented paradigm initiated by Kristen Nygaard and Ole-Johan Dahl, and further developed by Bertrand Meyer. Thus, reusable software modules became more than the resulting product of structural system decomposition: they became loosely coupled, highly cohesive conceptual units [63]. Under the strong influence of object-oriented technology, one of the first definitions of software components was given by Booch in [19]:

> A reusable software component is a logically cohesive, loosely coupled module that denotes a single abstraction.

The seminal work of Szyperski [77] further developed this concept, and the initial definition of a software component evolved to a richer view, with a strong accent on reusability. The loosely coupled perspective was thus taken further, towards a possible independence of development and deployment for various components

of the same system, as defined in [77]:

> A software component is a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

In [46, 39], the authors identify more types of reusable components that may appear in a component-based software project: commercial off-the-shelf components, which are independently deployable, in-house built reusable modules and libraries, with a rather limited degree of independence and reusability, and legacy modules (which can and have to be integrated into newer systems). What usually distinguishes the off-the-shelf components from software modules is their extended reusability and replaceability. Their implementation is usually hidden from the customers, and they are delivered as black boxes with documented interfaces, which only interact through their interfaces, without exposing their inner state [75]. Further on, based on their designated purpose, off-the-shelf components are also classified in individual, standalone reusable components and composite building blocks, which are specifically developed as building parts and conform to certain composition standards [39].

Composing new systems out of reusable, off-the-shelf components is clearly a more than promising direction in both current software engineering research and practice. As pointed out by Sommerville in [75] software reuse not only accelerates the development process, while reducing its risks, but also increases the dependability of the resulting product – as an already tried and tested component is definitely more reliable to use than a freshly developed one. Even more, as off-the-self components exhibit a coarser granularity, and tend to provide a lot of functionality, reusing them can lead to further important cuts to both development costs and time [75].

Increased development efficiency, safer and more reliable systems, great component versatility and reusability, all these would be easily achievable if software components would be as standard as lego pieces. It would make the process of building correct software as simple as building lego houses. Unfortunately, practical realities are much harsher and utterly complex. Thus, even though the future of software engineering lies in composability and reusability, there is often the case for the lego pieces to fit badly or not at all.

It is important to note that in a component-based paradigm the software development process is not only shorter than normally, but also more complex,

although it might actually appear simpler at a superficial look. First of all there are not one, but two different development processes, that translate to two different, overlapping software life-cycles: the component life-cycle, and the product life-cycle [27, 28]. In the system development process, the phases of unit design, development and testing are replaced by new, specific phases: component selection (from a preexistent component pool), adaptation and specific component verification and testing (in isolation, in an assembly, after deployment in the system, etc.). Also, in the component life-cycle the need to allow for wide reusability affects all the development process phases, from requirements elicitation and component design, to testing and maintenance. Further more, components have to be individually found, evaluated, and sometimes updated and/or replaced [27, 28].

Let us add to this inherent complexity and parallelism in the software development process another generally observed fact. Due to independent, third-party development of components, mismatches that appear between the assembled parts will usually prove difficult to solve, as difficult as these building blocks are new or barely familiar to the developer. Therefore, one of the main challenges in component-based deployment of systems is how to preserve the gain in efficiency and productivity offered by the use of existing software component, without compromising it to endless fixing of mismatching problems.

As early as 1995, Garlan et al. were discussing the difficulties encountered while building a new application out of already existing parts [40]. The incompatibilities encountered resulted in two years of hard work to solve. The development process was slow due to the fact that the component dependencies and interactions were many and quite hard to understand. Hard challenges were raised by the different assumptions the original developers of the components had made regarding infrastructure, the control and the data model, the interaction protocols and the exchanged data, etc. All these have led to severe mismatchings.

To eliminate at least partially this kind of problems some common conventions for off-the-shelf components are needed. Not only it was much better for the reused components to be regarded as merely black boxes by their reuser, as earlier stated, but also certain information about their inner structure, requirements, etc. needed to be available for better understanding. Harrold et al is proposing in [45] a technique to make reusable components easier to test and analyze by third parties. In order to facilitate these software engineering “musts” it is indicated that the component provider should not only support a great deal of testing of his components as context-independently as possible, but should

also deliver with his components relevant summary information, represented in a standard notation, about the component. This information should include static or dynamic slices of the component code, a list of exceptions raised and their raising conditions, different functional constraints, dataflow information, etc. All these for the purpose that the component user would be able to better understand, integrate, analyze, test and debug his application.

Other common conventions started to emerge for the reusable building blocks. Due to the importance of having some uniform architectural view on the building blocks, that would allow both ease of assembly and replacement, such common conventions have finally coagulated as component models. This lead to a new definition of a reusable software component when regarded as a building block, definition given by Bachmann et al. in [38]:

> A component is:
>
> - an opaque implementation of functionality
> - subject to third-party composition
> - conformant with a component model

Although somehow narrowing the scope of software components, the criterion of conformance to a component model is aimed to bringing uniformity and standardization in the development process. Component models impose architectural constraints on building blocks, by describing standard ways of interacting in a component-based system [38].

One of the first steps towards standard component models was the invention of the reusable middleware, standardized first by the Object Management Group (OMG) as a means to support a uniform interaction of various distributed objects over the network [80]. Thus, Common Object Request Broker Architecture (CORBA) standard was established, together with the OMG IDL interface specification language. Then, Sun went beyond the concept of middleware, creating a reusable application framework with the introduction of Enterprise Java Beans (EJB) component model, specifically adapted for business applications and which became widely adopted in industry [13, 47]. This also led to the CORBA standard being enriched by the CORBA Component Model (CCM), which was a more general application framework than EJB, being language-independent, but failed to achieve the same success as EJB [47]. Microsoft developed its own standard, called Component Object Model (COM), with its later versions COM+ and Distributed Object Model (DCOM), aiming to allow that way for the development

of large scale component-based applications [37].

However, obtained reliable composite systems out of preexistent building blocks asked for richer, more complex component models and better frameworks. Such models had to allow for well-specified contracts, for a formal description of the component behaviour, for the expression of non-functional properties, etc. so that powerful tools for the verification and validation of composite systems could be developed. Under this purpose, several component models using rich formalisms and being able to express various component and architecture properties have been developed in academis. Such are Fractal [20] and SOFA [21] that describe composite architectures in a hierarchical way, allow for the formally specifying the behaviour of both primitive and composed components, and use off-the-shelf model checking tools for verification. The Java/A component model integrates architectural and behavioural description by embedding specific information in the actual implementation, to avoid architectural erosion [8]. A formal model based on timed infinite streams, the Focus/Autofocus component model is well suited for reactive systems, and its aim is to provided a top-down guided development, by successive refinements, from given requirements to implementation [49]. The rCOS component model uses a relational calculus for an object-oriented analysis of the component based system, which can be described at various levels of abstraction [56], while the CoIn model captures the behaviour of the components in the system by means of component-interaction automata, and composition correctness is then verified by the DiVinE model checker [10]. The increased interest in formally enriched component models has lead to a component modeling challenge, employing a common, complex real-world case study for a uniform evaluation of many emerging approaches [3]. However, none of the mentioned formal component models are widely adopted in industry, as they (and their associated tools) still represent ongoing research.

Beyond common conventions and modeling perspectives, for an off-the-shelf component to be reused in a new system one must note that adaptation is required to reconcile the different, imperfectly matching interfaces [75]. In order for the adaptation to be possible, knowledge of the component inner dynamics needs to be available. Modern approaches to component adaptation, as surveyed in [22] and in [73] regard components as grey-boxes, where the knowledge needed for adaptation describes mainly the component behaviour and interaction protocols and is encoded by means of Finite State Machines, Labeled Transition Systems, Petri Nets, process algebra, etc., which facilitate formal reasoning about their behaviour in the context of the desired system. The general aim is to obtain an automated component adaptation within the new system.

2.2 Model-Based Adaptation

The component-based approach in software engineering is set on reusing already existing components in order to build new systems. Assembling these components into an application, or reassembling them in an attempt to add new features to the system is not trivial. Important issues raised by this paradigm relate to frequent component incompatibility. Forcing them to match the new schema may require a lot of development effort and can eventually end up with obtaining a fragile, badly integrated system.

Therefore, a lot of effort has been put into automatically adapting component behaviour to one another and also to the system's requirements. As the implementation details components are bound to remain hidden, the adaptation process has to happen in a non-intrusive ways.

There are currently three main approach typeses in component adaptation: particular, generative and restrictive [23]. Particular approaches refer to practical ad-hoc solutions which are not subject to generalisation. Generative approaches build mediators to rearrange communication protocols in order to obtain a good match. Also, restrictive solutions adapt component behaviour by excluding that part of it that would lead to integration problems. As the last two solutions are complementary, a hybrid approach, that would use a combination of generated adapters and pruning of mismatching behaviour has also appeared [23].

In 1997, Yellin and Strom have set the foundation for automatic model-based adaptation of reusable component behaviour [84]. They found a generative solution to the issue of incompatible component integration and this solution was to build and use adaptors. An adaptor was formally defined as a software entity which could enable correct interoperability of components with mismatching behaviour. They used finite state machines for modeling component interactions and defining compatibility relations among them. This allowed them to generate the needed adaptors in a semi-automated way [84].

Based on Yellin and Storm's work, Reussner and Schimdt have developed a rather ad-hoc than generative approach [71], using finite state machines (FSMs) to formally describe component behaviour. They assume that component configurations, with provided and required interfaces are also already known when composing a system. The terms of ken and gate are introduced to describe system architecture. A ken is a self-contained component, encompassing a cluster of internal objects, while a gate is an interface object, through which data or

control is transfered to or from kens. The gates are further divided into required and provided gates, where both required and provided behaviour is described by FSMs. Using a a design-by-contract view, the compatibility between two kens can be reduced to the compatibility between their required and provided gates. Several frequent component interaction and mismatch patterns are considered and specific adaptor-generating methods are developed for each such pattern [71].

This pattern-based method for adapting component and ensuring interoperability is later extended by Reussner with more restrictive aspects [68, 69]. The concept of parameterized contracts, as a generalisation of component interoperability checks is introduced. Partial compatibility between components becomes possible, and a component provided or required interface can be restricted in the adaptation process, so that the component can adapt to a large range of environments. A component will provide less functionality, if not all its required functionality is provided by the environment; also a component will require less functionality if not all its provided functionality is used by its clients. This significantly enhances the degree of component reusability [68, 69].

An interesting hybrid generative-restrictive approach is the one proposed in 2008 by C. Canal, P. Poizat and G. Salaun [23]. While sharing some similarities with the Reussner et al. method, Canal et al. offer a more general solution, adapted to complex scenario situations and have a well developed formal notation for specifying the adaptation contracts. The component behaviour is modeled using Labeled Transitions Systems (LTS), and, to better express communication protocols, labels used are vectors. Two adaptation algorithms are proposed. The first one performs a restrictive adaptation, using an extension of the synchronous product, synchronous vector product, which is computed between the adaptation contract LTS vector and the set of LTS vectors corresponding to the system components. Out of the result LTS, all paths leading to deadlock situations are eliminated. The second algorithm uses Petri Nets to allow for the reordering of messages passed between components. For this purpose, a Petri net is generated for every component protocol, starting from its LTS vectors. The adaptation contract is, in this case, also issued in the form of Petri nets. [23].

An interesting generative solution is the one of Brogi et al., which uses a formal specification method known as synchronous π–calculus based on representing components through roles or patterns. An adapter specification would be described by a mapping establishing some rules that describe relations between the actions and the data of involved components. This specification is refined by

an adaptor generating algorithm and a concrete implementation of the adaptor is obtained. Same mapping may later be used to formally verify whether the generated adaptor is correct [17, 18].

A notable restrictive approach is developed by Inverardi and Tivoli [6, 5, 78]. Their work is based on a specific architecture, containing specific coordinator components, also known as connectors, and component behaviour is modeled as LTS. A coordinator is generated for a set of components and a desired coordination policy, which expresses the set of allowed behaviours in the system, by computing a synchronous product between participating LTS and by removing the paths to deadlock. Also, a two step method is offered for enhancing existing communication protocols and adding new features, by applying several local modifications on the coordinator components and generating some wrapping "glue code" [6]. For the case of distributed software components, a special distributed solution allows for localized adaptation, by first obtaining a centralized coordinator, then deriving local behavioural filters from its projections on the local component domains, and keeping only a small central coordinator to avoid system deadlocks [5, 78].

Another restrictive, very influential solution belongs to de Alfaro and Henzinger who propose an optimistic definition of compatibility: two components are compatible if there exists an environment in which they interoperate adequately. They present a light-weighted formalism that captures also the temporal aspect of component interfaces. This is achieved by using an automata-based language, interface automata, to capture input assumptions about the invoking order of component methods, and also output guarantees of the order in which the component is invoking external methods. He obtains component behavioural adaptation using a game theory-inspired alternating approach, under the assumption that one interface refines, and thus is compatible with another if it has weaker input assumptions and stronger output guarantees [31].

Presented approaches generally consider synchronous and/or completely controllable components. However, asynchronous components, which interact by sending and receiving messages, can be only partially controllabler, since it's possible that a message send event cannot be forced from outside. In this case, correct composition becomes a control problem [67, 7], where the controllable events are the message receive events, while message sending events are considered uncontrollable from an external point of view. In this case, an adaptor/coordinator for the desired system is obtained from a generated controller, which restricts system behaviour to the behaviour allowed by a desired property,

thus this approach can too be consired restrictive [67, 7].

Due to the increasing practical interest in reusing and recomposing existing software components, automated software adaptation and composition by adaptor/coordinator generation has received a lot of attention. However, all formal adaptation techniques require a formal specification of component behaviour. If formal behavioural specifications are an important part of some of the most recent component models from academia [3], providing them is still far from being common industrial practice. Thus, when building a system, it if often the case of having to integrate partially specified components. To be able to automatically compose such a system, in a provably correct way, one has to first extract the necessary knowledge on component behaviour, in a formal way. This is done by model learning.

2.3 Finite State Machines

A *Finite State Machine* (FSM) is a mathematical model used to describe systems that can be in one of a finite number of *states* at a time. The system can pass from one state to another when relevant events occur, and this is considered a *transition*. The relevant events that trigger such transition form the *alphabet* of the finite state machine. In the beginning, the system is in a state known as the initial state. When the system has executed a sequence of transitions that represents a meaningful execution scenario, the system reaches an *accepting* state, also known as a *final* state. A finite state machine can have one or several accepting states.

A finite state machine is *deterministic*, when from each state no more than one transition on the same event can be taken, and is *nondeterministic* when, from at least one state, more than one transition can be taken on the same event. An example of a deterministic finite state machine can be seen in figure 2.1.

Formally, a finite state machine can be defined as a quintuple:

$$M = \langle Q, q_0, Q_f, \Sigma, \delta \rangle$$

where:

- Q is the set of states
- q_0 is the initial state

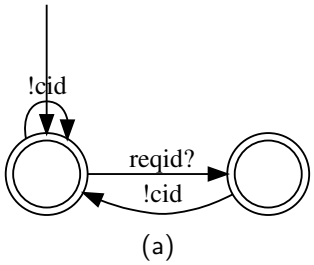


Figure 2.1: A deterministic finite state machine.

- Q_f is the set of final states
- Σ is the alphabet of events
- δ is the partial transition function and is defined as $\delta : Q \times \Sigma \rightarrow Q$ for deterministic finite state machines, or as $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$ for nondeterministic finite state machines

If, for a deterministic finite state machine, the function δ is defined for state $q \in Q$ and event $\sigma \in \Sigma$, we denote this by $\exists\delta(q, \sigma)$.

A deterministic finite state machine is also known under the name of deterministic finite automaton (DFA).

Considering the set of events Σ, a *word* is a finite string of events in Σ. The set of all finite strings composed from events in Σ is denoted as Σ_*. Let $\delta_* : \Sigma_* \rightarrow Q$ be the extension of the partial transition function δ to words in Σ: $\delta_*(\sigma) = \delta(q_0, \sigma)$, and $\delta_*(w\sigma) = \delta(\delta_*(w), \sigma)$, where $w \in \Sigma_*$.

If a word w in Σ_* triggers a sequence of transitions that takes the automaton, step by step, from the initial state to an accepting state: $\delta_*(w) \in Q_f$, we say that the word w is accepted by the automaton.

A subset of the string set Σ_* is a *language*. A language $L \subseteq \Sigma_*$ is a regular language if a deterministic finite automaton $A = \langle Q, q_0, Q_f, \Sigma, \delta\rangle$ exists, so that A will accept all the strings in L and only them $L = L(A)$, where $L(A) = \{w \in \Sigma_* | \delta_*(w) \in Q_f\}$. For a language L can exist more automata A_i, so that $L = L(A_i)$. However, there exists an automaton A_j, $L = L(A_j)$, so that A_j has the minimum set of states from all automata accepting L.

Let us consider two automata A_1 and A_2. We say they are synchronized when

both automata take a transition on a common event at the same time. Then, their synchronous product is the automaton $A_1 \| A_2 = \langle Q_{12}, q_{012}, Q_{f12}, \Sigma_{12}, \delta \rangle$, where $Q_{12} = Q_1 \times Q_2$, the initial state is $q_{012} = (q_{01}, q_{02})$, the set of accepting states is $Q_{f12} = Q_{f1} \times Q_{f2}$, the event set is $\Sigma_{12} = \Sigma_1 \cup \Sigma_2$ and the transition function $\delta : (Q_{12}) \times (\Sigma_{12}) \to Q_{12}$ is defined as following:

- $\delta((q_1, q_2), \sigma) = (q_1', q_2')$ if $\delta_1(q_1, \sigma) = q_1'$ and $\delta_2(q_2, \sigma) = q_2'$
- $\delta((q_1, q_2), \sigma) = (q_1', q_2)$ if $\delta_1(q_1, \sigma) = q_1'$ and $!\exists \delta_2(q_2, \sigma)$
- $\delta((q_1, q_2), \sigma) = (q_1, q_2')$ if $\delta_2(q_2, \sigma) = q_2'$ and $!\exists \delta_1(q_1, \sigma)$

When only one of the automata takes a transition on a common event, i.e. the automata do no synchronize, their composed language is given by the asynchronous product. The asynchronous product of two automata A_1 and A_2 is the automaton $A_1 \times A_2 = \langle Q_{12}, q_{012}, Q_{f12}, \Sigma_{12}, \delta \rangle$. The state set is $Q_{12} = Q_1 \times Q_2$, with the initial state (q_{01}, q_{02}) and the set of accepting states $Q_{f12} = Q_{f1} \times Q_{f2}$. The event set is $\Sigma_{12} = \Sigma_1 \cup \Sigma_2$. The transition function for the asynchronous product is defined as:

- $\delta((q_1, q_2), \sigma) = (q_1', q_2)$ if $\sigma \in \Sigma_1$ and $\delta_1(q_1, \sigma) = q_1'$
- $\delta((q_1, q_2), \sigma) = (q_1, q_2')$ if $\sigma \in \Sigma_2$ and $\delta_2(q_2, \sigma) = q_2'$

As more specialized types of finite state machines, we will further introduce Mealy machines, interface automata and input/output automata.

A *Mealy machine* is a finite state machine with input/output events. It is formally described by a 7-tuple $\langle Q, q_0, \Sigma, \Lambda, Q_f, \delta, \eta \rangle$, where while Σ is the set of input events, Λ is the set of outputs, and a transition in the automaton both consumes and input event and produces an output event. Thus, a transition is both defined by the transition function $\delta : Q \times \Sigma \to Q$, and by the output function $\eta : Q \times \Sigma \to \Lambda$, which associates a pair of a state and an input symbol to an output symbol.

An *interface automaton* is a finite state machine designed to describe communicating systems, having disjoint sets of input and output events. It is described by a tuple $\langle Q, q_0, \Sigma, \Lambda, Q_f, \delta \rangle$, where Σ is the set of input events, Λ is the set of output events, and the transition function is defined as $\delta : Q \times (\Sigma \cup \Lambda) \to Q$, thus a transition in the interface automaton can be triggered from the current state by either an input, or an output event. Finally, an *input/output automaton* is an interface automaton in which each input event is enabled in any state.

Let us now further consider the notion of *determinism*. Conceptually, a system

is deterministic when for each string of input events it consumes, it produces a unique string of output events. This is also known as *behaviour determinism*. A deterministic finite automaton, as well as a deterministic Mealy machine are behaviour deterministic, since their next transition and/or output event is uniquely determined by the input event provided. However, for finite state machines that have both input and output events, such as input automata, the issue is more complex.

An interface automaton is *input deterministic* if only one transition on an input event can exist from any state $q \in Q$, so $(\delta(q, \sigma) = q_1) \wedge (\delta(q, \sigma) = q_2) \rightarrow q_1 = q_2$, where $\sigma \in \Sigma$. An interface automaton is *output deterministic* if for each output event, no more than one transition can occur from a state q, therefore $(\delta(q, \lambda) = q_1) \wedge (\delta(q, \lambda) = q_2) \rightarrow q_1 = q_2$, for $\lambda \in \Lambda$. Also, an interface automaton is *output determined* if no more than one transition on an output event can occur from a current state, thus: $(\delta(q, \lambda_1) = q_1) \wedge (\delta(q, \lambda_2) = q_2) \rightarrow q_1 = q_2 \wedge \lambda_1 = \lambda_2$. An I/O automaton is behaviour deterministic if it is both input deterministic and output determined.

2.4 Model Learning

Learning an unknown language, also known as the regular inference problem, means finding the deterministic finite automaton that describes that regular language. Various methods have been developed to address this problem, some of them based on passive learning, i.e. learning from a set of available observed traces, which represent either only positive samples, or both positive and negative samples, and some of them considering active learning, where the samples are obtained at runtime, by querying, whenever more knowledge is necessary. The best known active learning technique is the L* algorithm, also known as the Angluin algorithm, developed by Angluin [4] for deterministic finite-state automata and many black box model inference methods are based on it.

2.4.1 The L* algorithm

Introduction

As introduced in [4], the L* algorithm learns a deterministic finite state automaton under the following assumptions:

- the alphabet Σ of the automaton is known
- a bound n on the number of states of the automaton is also known
- the machine can be reset anytime to its initial state

The algorithm learns a model by repeatedly asking queries, and by constructing an automaton based on the resulting observations. Two types of queries can be asked: membership queries, that consist in asking whether a given string $s \in \Sigma_*$ is accepted by the unknown automaton, and equivalence queries, which are asked once a conjecture automaton has been obtained. Membership queries are iteratively asked. The answers to the membership queries are considered observations, and are used to generate a conjecture automaton when certain criteria are met. For equivalence queries, the algorithm assumes the existence of an oracle, which knows the real automaton to be learned. The oracle answers an equivalence query either by confirming the equivalence of the learned and real automata, or, if the learned automaton is not equivalent to the real one, by providing a counterexample trace, that either is accepted by the real automaton and rejected by the conjecture, or is accepted by the conjecture and rejected by the target [4]. In practice, however, such an oracle does not exist, and therefore it is emulated by an often large set of membership queries.

Learning a regular language

As described in [4] the learning takes place in the following way. An observation table is permanently maintained, recording the answers to the membership queries. Let S and E be two sets of strings in Σ_*, where S is a prefix-closed set, and E a suffix-closed set of strings. The observation table O is a two-dimensional array, with rows labeled by the elements in $S \cup S\Sigma$, and the columns labeled by elements in E. A string $s \in S$ represents the shortest prefix to access a state, while a string $e \in E$ represents a differentiating suffix. The strings s in string set S, and the strings e in string set E are obtained from the membership queries, with s always being a prefix and e a suffix of a same query. An entry in the observation table, $O[s, e]$, is 1 if se is accepted by the unknown automaton, and 0 otherwise. In the beginning, both S and E only contain the empty string ϵ, thus the learning starts with the hypothesis of the trivial automaton.

Two rows in the observation table, corresponding to strings s and t are considered equivalent if for all $e \in E$, the entries $O[s, e] = O[t, e]$. The observation table is considered closed if for each $t \in S\Sigma$ there is an $s \in S$, so their corresponding

rows are equivalent. Also, the observation table is considered consistent if for each strings s and t in S, whose rows in the observation table are equivalent, for all $\sigma \in \Sigma$, the strings $s\sigma$ and $t\sigma$ have equivalent rows in the table. While the observation table is not both closed and consistent, membership queries are asked. First, the observation table is filled in the following way. For each string $s \in S \cup S\Sigma$ and $e \in E$, a membership query for se is issued, and the result of the query is recorded in the corresponding observation table entry [4].

If, after the table is filled, it is not closed, then the trace $t \in S\Sigma$ for which no equivalent $s \in S$ exists is moved to S, and a new set of queries te is asked for all $e \in E$, to fill the added row in the table. If the observation table is not consistent, then equivalent strings s and t in S, for which $s\sigma$ an $t\sigma$ are not equivalent, are found, and string σe is added to E, thus becoming a differentiating prefix. Then, a new set of queries is asked to fill the added column for all rows in the table [4].

When an observation table is both closed and consistent, a conjecture automaton can be built from it.

The conjecture automaton is constructed in the following way. The strings in s are the access prefixes, as they represent the shortest prefix by which a state can be accessed, thus each prefix in S can be associated to a state of the conjecture automaton. The strings in $S \cup S\Sigma$ are used to construct the transition function, so that for each state corresponding to a string in S, a transition on an event $\sigma \in \Sigma$ exists. The obtained conjecture automaton is minimal [4].

Then, an equivalence query is asked to the oracle, in order to determine whether the conjecture automaton is equivalent to the unknown one. If this is true, the oracle answers affirmatively. Otherwise, it provides a counterexample. If a counterexample is provided, the observation table is extended by adding all the prefixes of the counterexamples to S. The newly added rows in the table are then filled by asking new membership queries. Afterwards, the table is checked for, and eventually made closed and consistent, and a new conjecture automaton is obtained. The learning process ends when the equivalence query returns true, and thus the hypothesis model is confirmed [4].

Several optimisations have been proposed for counterexample processing, such as [70], [61], [74], in order to reduce the number of new membership queries. The method of Rivest and Shapire analyzes counterexamples separately and only adds one counterexample suffix to the set of distinguishing sequences E, while still adding all counterexample prefixes to S [70] – this helps in saving membership

queries, but the resulting conjecture automata might not be minimal anymore. Maler and Pnueli process the counterexample by adding all its suffixes to set E [61], while Shahbaz removes from the counterexample its longest prefix in $S \cup S\Sigma$ – its longest already tried prefix – and only adds all suffixes of the remaining sequence to E [74].

Another kind of optimisation, aiming to reduce the number of membership queries, was developed by Kearns and Vazirani. They split the observation table into subtables by using a discrimination tree. The counterexample suffixes are only added locally, to one subtable. This results in important savings on the side of membership queries, but might also lead to an increase in the number of necessary equivalence queries [58].

Equivalence queries

In practice, the oracle that answers equivalence queries does not exist, it is purely a hypothetical construction that allows for an elegant description of the L* algorithm. Therefore, when learning black box components, an equivalence query has to be approximated by an exhaustive set of membership queries. If no maximum state assumption would have been made on the unknown automaton, an equivalence query would have been undecidable. However, with the automaton size bounded by a maximum value m, an equivalence query can simply consist of an exhaustive testing of the learned language via membership queries.

A more efficient solution, however, is using an algorithm for conformance testing, such as the one developed by Vasilevskii and Chow [79, 26]. By using conformance testing, instead of having an equivalence query that is exponential in the maximum size of the unknown system, an equivalence query exponential only in the difference between the size of the learned automaton and its maximum bound is obtained. Further on, the equivalence query generation technique described by Howar et al. in [55], also known as the "evolving hypothesis" is able to find counterexamples fast when they exist, thus highly reducing the number of membership queries asked for answering one equivalence query.

2.5 Control of Discrete Event Systems

As defined in [67], a Discrete Event System (DES) is a dynamic system whose evolution is determined by the occurrence of events from a certain event set. If we associate the discrete events from the event set of a specific DES to the distinct symbols of a finite set Σ, we can then refer to Σ as the alphabet for that DES. The symbols in Σ will further be known as event labels.

Then, the behaviour of the discrete event system can be formally described by a language over Σ. As shown in [81], and using the authors' terminology, this language can be conveniently expressed as a generator:

$$G = \langle Q, \Sigma, \delta, q_0, Q_m \rangle$$

where Q is the countable state set, $\delta : Q \times \Sigma \to Q$ is the partial transition function, state $q_0 \in Q$ is the initial state of the DES, and the set of states $Q_m \subseteq Q$ is the subset of marked states. Marked states can somehow be associated to acceptance states, which they semantically resemble, although a clear distinction is made between the language of G and the marked language of G, a sublanguage of the first. Assume that $\epsilon \in \Sigma$, where ϵ is the empty event.

Let Σ_* be the set of all possible sequences s over Σ, and let $\delta_* : Q \times \Sigma_* \to Q$ be the extension of δ over Σ_*, so that $\delta_*(q, \sigma) = \delta(q, \sigma)$ and $\delta_*(q, \sigma.s) = \delta(\delta(q, \sigma), s)$, where $\sigma \in \Sigma$.

The language $\mathcal{L}(G)$ consisting of a subset of event sequences in Σ_* so that $\delta_*(q_0, s)$ is defined for all sequences $s \in \mathcal{L}(G)$ represents the *closed behaviour* of G. Also, the subset of sequences $s \in \mathcal{L}(G)$ for which $\delta_*(q_0, s) \in Q_m$ represent the *marked behaviour* of G, noted by $\mathcal{L}_m(G)$ [81].

A state $q \in Q$ is considered reachable if a string $s \in \mathcal{L}(G)$ exists such that $\delta_*(q_0, s) = q$, and the generator G is *reachable* if all its states $q \in Q$ are reachable. Also, a state q is considered *coreachable* if a string $s \in \Sigma_*$ exists so that $\delta_*(q, s) \in Q_m$, that is if a marked state can be reached from q, and G is *coreachable* if all its states $q \in Q$ are coreachable. If every reachable state of G is coreachable, then G is *nonblocking*, thus every string s in $\mathcal{L}(G)$ is a prefix of a string in $\mathcal{L}_m(G)$, $\mathcal{L}(G) = \overline{\mathcal{L}_m(G)}$, where $\overline{\mathcal{L}_m(G)}$ is the prefix-closure of the language $\mathcal{L}_m(G)$. If a generator G is both reachable and coreachable, then the generator G is *trim* [81].

Further on, we shall define the synchronous product of two languages $L_1 \subseteq \Sigma_*^1$ and $L_2 \subseteq \Sigma_*^2$ corresponding to two different discrete event systems DES_1 and DES_2. Let us assume $\Sigma^1 \cap \Sigma^2 \neq \emptyset$. Also, the reunion of the two event sets is $\Sigma^1 \cup \Sigma^2 = \Sigma$.

Let us have a function $P_i : \Sigma_* \to \Sigma_*^i$, defined as following:

- $P_i(\epsilon) = \epsilon$
- $P_i(\sigma) = \begin{cases} \epsilon, & \sigma \notin \Sigma^i \\ \sigma, & \sigma \in \Sigma^i \end{cases}$
- $P_i(s.\sigma) = P_i(s).P_i(\sigma), \ \forall s \in \Sigma_*, \sigma \in \Sigma$

The authors define P_i as the *natural projection* of sequence set Σ_* onto Σ_*^i [67]. The inverse of P_i, $P_i^{-1} : Pwr(\Sigma_*^i) \to Pwr(\Sigma_*)$, where $Pwr(M)$ is the powerset of set M, is then defined as:

$$P_i^{-1}(H) = \{s \in \Sigma_* | P_i(s) \in H\}, \ \forall H \in \Sigma_*^i$$

For languages $L_1 \subseteq \Sigma_*^1$ and $L_2 \subseteq \Sigma_*^2$, the *synchronous product* $L_1 \| L_2$ is:

$$L_1 \| L_2 = P_1^{-1}(L_1) \cap P_2^{-1}(L_2)$$

Thus, for two generators G_1 and G_2, the synchronous product of their languages is a generator G so that $\mathcal{L}_m(G) = \mathcal{L}_m(G_1) \| \mathcal{L}_m(G_2)$, and $\mathcal{L}(G) = \mathcal{L}(G_1) \| \mathcal{L}(G_2)$.

According to [67], the events that determine the evolution of the discrete event system are of two kinds:

- controllable events: events that can be enabled or disabled by an agent external to the DES; transitions on controllable events are taken only if that controllable event is enabled, otherwise the event doesn't occur and the transition cannot be taken
- uncontrollable events: events that cannot be enabled or disabled by an external agent, they occur independently of external control, and transitions labeled by such events are taken based on inner, nondeterministic decisions of the DES

Thus, the alphabet Σ contains both symbols that label controllable events, and symbols that label uncontrollable events: $\Sigma = \Sigma_c + \Sigma_u$.

A *control pattern* is a subset of the controllable events, together with all the uncontrollable events of the discrete event system. Let the set of all control patterns be $\Gamma = \{\gamma \in Pwr(\Sigma) | \gamma \supseteq \Sigma_u\}$.

Then, a *supervisory control* for the generator G is any map $V : \mathcal{L}(G) \rightarrow \Gamma$, and V/G represents the generator G under the supervision of V. The language $\mathcal{L}(V/G) \subseteq \mathcal{L}(G)$ is the *closed behaviour* of V/G, where:

- $\epsilon \in \mathcal{L}(V/G)$
- $s \in \mathcal{L}(V/G) \ \wedge \ \sigma \in V(s) \ \wedge \ s.\sigma \in \mathcal{L}(G)$ implies $s.\sigma \in \mathcal{L}(V/G)$
- no other strings belong to $\mathcal{L}(V/G)$

Also, the *marked behaviour* of V/G is $\mathcal{L}_m(V/G) = \mathcal{L}(V/G) \cap \mathcal{L}_m(G)$. If $\overline{\mathcal{L}_m(V/G)} = \mathcal{L}(V/G)$, i.e. if the language described by V/G is equivalent to the prefix-closure of its marked language, then V is nonblocking for G. For a language $\mathcal{M} \subseteq \mathcal{L}_m(G)$, a *marking nonblocking supervisory control* for (M, G) is a supervisory control for which $\overline{\mathcal{L}_m(V/G)} = \mathcal{L}(V/G) \cap M$ [81].

A language $K \subseteq \Sigma_*$ is controllable with respect to G iff

$$\overline{K}\Sigma_u \cap \mathcal{L}(G) \subseteq \overline{K}$$

This means that the occurence of any uncontrollable events after a prefix of a sequence in K leads to a trace that, if in $\mathcal{L}(G)$, is still a prefix of a sequence in K. Thus, if K describes a desired behaviour, the occurence of uncontrollable events cannot lead to erroneous executions that are also in $\mathcal{L}(G)$.

Then, according to [81], if $K \subseteq \mathcal{L}_m(G)$ and $K \neq \emptyset$, then a *marking nonblocking supervisory control* V for (K, G) such that $\mathcal{L}_m(V/G) = K$ exists if and only if K is controllable with respect to G [81].

This marking nonblocking supervisory control can be implemented by an automaton S such that $K = \mathcal{L}_m(S) \cap \mathcal{L}_m(G)$ and $\overline{K} = \mathcal{L}(S) \cap \mathcal{L}(G)$. In this case, S is a supervisor for G. This means that after a sequence $s \in \overline{K}$, a controllable event σ is only enabled if $s\sigma \in \mathcal{L}(S)$, so that only sequences in $\overline{K}$ are obtained, no matter what uncontrollable events get to occur. A nonblocking supervisor ensures that, if new events keep occuring, a sequence in K will eventually be obtained.

Let S' be a generator such that $\mathcal{L}(S') \in \Sigma$. Then, S' is a proper supervisor for G iff $\mathcal{L}_m(S')$ is controllable with respect to G, S' is trim and $\mathcal{L}_m(S')$ and $\mathcal{L}_m(G)$ are nonconflicting, i.e. $\overline{\mathcal{L}_m(S') \cap \mathcal{L}_m(G)} = \overline{\mathcal{L}_m(S')} \cap \overline{\mathcal{L}_m(G)}$ [81].

Let there be a language $E \subseteq \Sigma_*$. Then, the set of all sublanguages of E that are controllable with respect to G is $\mathcal{C}(E) = \{K \subseteq E|$ K is controllable with respect to G $\}$. In [81] it is proved that $\mathcal{C}(E)$ always contains a unique supremal element, which is denoted by $sup\mathcal{C}(E)$.

So, let us assume the language $E \subseteq \Sigma_*$ as an upper bound on an admissible marked behaviour, i.e. the specification language, with the property that $E = \mathcal{L}_m$(E), where E is the generator corresponding to E.

We want to obtain a minimally restrictive proper supervisor S for G such as $\mathcal{L}_m(S/G) \subseteq E$. Since it is minimally restrictive, S should allow for as much behaviour as possible, thus it is exactly the generator for the unique supremal controllable sublanguage of $E \cap \mathcal{L}_m(G)$. This supervisor can then be computed as a trim generator for the language $K = sup\mathcal{C}(E \cap \mathcal{L}_m(G))$ by using the following fixpoint procedure.

Assume the operator $\Omega : Pwr(\Sigma_*) \rightarrow Pwr(\Sigma_*)$ where $\Omega(Z) = E \cap \mathcal{L}_m(G) \cap sup\{T \subseteq \Sigma_* | T = \overline{T},\ T\Sigma_u \cap \mathcal{L}(G) \subseteq \overline{Z}\}$. Thus, for a language Z, Ω returns a language that contains only those traces in $E \cap \mathcal{L}_m(G)$ that belong to the largest prefix-closed language T whose extension by any uncontrollable event, is in the prefix-closure of Z if it also is in $\mathcal{L}(G)$. This leads to traces which, if suffixed by some uncontrollable event, lead to some new traces not in Z, being eliminated from $\Omega(Z)$, where $\Omega(Z)$ is also a sublanguage of language $E \cap \mathcal{L}_m(G)$.

Considering the desired language K from above, $K = sup\mathcal{C}(E \cap \mathcal{L}_m(G))$, K is the largest fixpoint of Ω, and it can be computed by iterative approximations. If $K_0 = E \cap\ L_m(G)$ and $K_{j+1} = \Omega(K_j)$, then $K = \lim\ K_j$ for $j \rightarrow \infty$, and this limit is reached after a finite number of steps [81]. Thus, the desired supervisor S that enforces E over G in a minimally restrictive way is obtained as the generator for language $K = sup\mathcal{C}(E \cap \mathcal{L}_m(G))$ computed by the largest fixpoint procedure.

Chapter 3

Related Work

3.1 Black Box Model Checking

A classical reference in the area of property-wise model learning for verification is the black box checking technique developed by Peled et al. for programs with missing or inaccurate models [33], using the Angluin algorithm [4] for model inference. The hypothesis model is proposed, verified, and compared to the real behaviour using black box testing. Found differences are used to generate a new hypothesis model, while counterexamples obtained from model checking are confirmed or invalidated by testing. The aim of the technique is to obtain a confirmed counterexample for the desired temporal property. In contrast to our approach, their model only has controllable input events, so all considered execution traces are completely controllable. The learning stops once the confirmed counterexample is obtained, without going all the way towards a complete model. This has inspired our run-time behaviour exploration technique, that also focuses on behaviour relevant to a desired property. However, the models we learn are meant for composition purposes, so, although traces violating the specification might be observed, the learning continues within some pre-established cost limits or until obtaining a precise model.

Xie and Dang propose in [82] a technique for CTL model checking of systems containing an unspecified component. Their algorithm gives a sufficient and necessary condition for the unspecified component as a witness graph, in order to ensure that the system will satisfy the desired property. Test sequences are generated using the witness graph. While also using jointly verification and

black box testing, our approach is focused on inferring an approximated model of the unknown component, such that this model can be appropriately used for system composition. Also, our technique deals with the more complex issue of controllability when exploring the system, and it is also flexible in the number of unspecified components it can address.

3.2 Black Box Model Learning – Angluin-Based Methods

While being the first to adapt the Angluin algorithm to Mealy machines, Niese also introduces in [64] the idea of using specific query filters (like for prefix-closed languages, or for partial order reduction), that highly improves the efficiency of the learning process using the Angluin algorithm for input-enabled components, by significantly reducing the number of queries. Since the languages we learn are also prefix-closed, our online inference algorithm indirectly uses a form of the prefix-closed language filter, by removing from the model the shortest prefix of a negative sample, however, our approach performs a step-by-step, depth-first exploration of the black box behaviour, thus it will always start with a shortest prefix to which confirmed events are iteratively appended.

The regular inference method of Berg et al. [11, 14] uses an adaptation of the Angluin algorithm [4], thus assuming the possibility of querying for trace membership and model equivalence. This approach can also deal with infinite state and even infinite alphabet components, by making use of symbolic representation and predicate abstraction techniques.

Apart from the work of Berg, another noteworthy learning approach derived from the Angluin's L* algorithm is RALT, the work of Shahbaz [74], which can infer parametrized Mealy machines by adding invariant learning to regular inference, and studies a component behaviour first in isolation and then while in interaction with the rest of the system. Other automata learning tools are LearnLib [65], which also learns Mealy machines, and Libalf [16], which learns deterministic and even nondeterministic automata – both are algorithm libraries, containing various optimizations and extensions of the classic Angluin algorithm. However, none of these approaches deals with the issue of uncontrollable events, which would make membership and equivalence queries difficult, as a specific query might require a large number of repeated tries to be answered.

Important advances towards addressing model inference for component with asynchronous messaging are done by Aarts and Vaandrager [2], which use a transducer to translate between I/O automata and Mealy machines, which are then learned using LearnLib. However, the I/O automata they address are still input-determined and do not exhibit any uncontrollable behaviour, having at most one output transition per state. Another common point to our method is that they learn models under a "learning purpose", which significantly reduces the number of necessary queries. Their technique is applied to case studies such as the biometric passport [1] or the Session Initiation Protocol (SIP).

The L* based approach of Bollig et al. [15] for learning residual nondeterministic finite state automata is considering nondeterminism in the model, however, the alphabet taken into account only has input events, thus nondeterminism, in this case, appears merely as a solution to obtain smaller inferred models.

El-Fakih et al. [34] have developed an adaptation of the Angluin algorithm in order to learn observable, nondeterministic Mealy machines. As Mealy machines are considered, the queries appear to be of two types: input and output queries. They acknowledge the need to repeatedly apply an output query in the context of nondeterminism, but the degree of nondeterminism (i.e., the average number of nondeterministic transitions from a state) they consider is very limited, being close to 1, where 1 corresponds to a deterministic machine. Thus, the impact of the nondeterminism on the overall cost of the model inference process is manageable, which would not be the case if the component had a larger number of nondeterministic transitions. Our algorithm does not place such restrictions on the degree of output-nondeterminism of a learned finite state machine, the number of uncontrollable transitions from one state can be as high as the number of uncontrollable events in the alphabet.

3.3 Model Learning and Controller Synthesis

Another closely related work, combining model learning and controller synthesis, is the approach of Hiraishi [48], that learns a supervisor for the system when the system specification, i.e., the desired property, is considered unknown, while the behavioural models for the components in the plant are provided. In this case, the membership and the equivalence queries are answered to by the system designer.

The white-box model inference method described in [42] by Păsăreanu et al.,

also combines model learning and controller synthesis. This work relies on environmental assumption generation when verifying a software component against a property. Their approach is based on the work of de Alfaro and Henzinger [31] stating that two components are compatible if there exists an environment that enables them to correctly work together. This divide-and-conquer technique analyzes components separately to obtain for each the weakest environment needed for the property to hold. By building a system controller, our approach also creates such an environment. However, while in [42] the analyzed component is well specified, our approach specifically addresses systems with black box components, whose behaviour and controllability must be understood before building an adaptor.

3.4 Non-Angluin Model Inference

Recent advances on dynamic model mining underline the critical significance of this domain. The GK-tail algorithm developed by Lorenzoli et al. [60] focuses on extracting extended finite state machines from execution traces – it is thus a passive learning algorithm. GK-tail uses inferred invariants and positive execution samples to extract EFSMs In contrast, our approach learns the model using both positive and negative samples, while always obtaining a safe approximation of the real component behaviour. Also, our samples are not random, but are generated using a property-driven strategy, thus our learned models are obtained specifically aiming towards a composition goal. Currently, our approach does not consider inferring guard conditions over the values of specific data parameters, but we consider this as a powerful and interesting direction for future work.

In [76], Suman et al. describe an active learning method to extract state models for black box components under the form of finite state machines with guard conditions. It considers that a state is defined by the method invocations it accepts, and it discovers potential new states by invoking all active methods from the current state. These potential states can be merged or confirmed, existing cycles can be detected within a certain bound, etc. Somehow similar, the work of Dallmeier et al. in [29] relies on test case generation to systematically extend the execution space and thus obtain a better behavioural model by dynamic specification mining. The execution space is extended by adding/removing method calls from the current state, and thus enriching an existing test suite. Also, in [41], Ghezzi et al. present the SPY approach, which infers behavioural models of black box components that embody data abstractions. All these tech-

niques are designed to work with synchronous method calls, while our approach addresses asynchronous message exchange, with uncontrollable send messages, which is more difficult as it involves the issue of uncontrollable events. Also, as already mentioned, our technique is composition-oriented, thus the component behaviour is explored not necessarily exhaustively, but only considering behaviour relevant to both the desired property and the feasible interactions with the other components in the system.

In the domain of conversational web services, we found the work of Bertolino et al. [12] on behaviour protocol synthesis, and Cavallaro et al. [24] on service adaptation to be related to ours. While [12] describes a technique that uses the WSDL interface to extract possible operation dependencies between the I/O data, and then validates these dependencies through testing, thus obtaining the behavioural web service protocol, the work in [24] takes this technique further, and develops a method for synthesizing web service adaptors starting from WSDL descriptions, which enables a correct interoperation when an old service is replaced by an equivalent, new one. However, the web services in [12, 24] are stateless, while our approach addresses stateful black box components. Thus, while their work starts from a data-dependency perspective, ours explores the language of the component and its controllability, while regarding data from a higher level of abstraction, as passed and received messages. Therefore, we consider the two approaches complementary.

3.5 Testing Asynchronous Black Boxes

In the area of testing black box components, Grabe et al. [43] develop a consistent and interesting technique for testing asynchronous black box components with uncontrollable send events, by using an executable interface specification. Their method addresses the difference between interacting under component control or under environment control, by describing the component behaviour in an assumption-commitment style using a rich formal notation. This work is closely related to ours, as we also test asynchronous black boxes – any runtime query is nevertheless a test – , and we also use a variant of an executable specification when we synchronize the property automaton with monitored components. However, their method doesn't infer a model for the tested black box component.

The problem of testing asynchronous black boxes for conformance is also ad-

dressed by Kaschner in [57], where stateful black box services with asynchronously sent messages that can overtake each other, using parallel communication channels, are considered. In order to appropriately react to uncontrollable send events, test cases are synthesized as specific partner services starting from the desired specification. This work does not infer any models for the tested services, aiming only to discover situations of non-conformance to the specification. However, considering parallel communication channels, and deducing the acceptance of a message indirectly are further developments that would also benefit our method.

3.6 Verification-Driven Execution

The use of verification-driven execution also relates our method to smart play-out [30], which is a lookahead technique that employs model checking to execute and analyze Live Sequence Charts. The play-out technique is mainly used to actually execute specifications from a GUI, during the software design process. Both approaches use verification-driven execution to improve knowledge, but, while we infer existing, unknown behaviour, smart play-out experiments with execution scenarios to find the best design options.

CrystalBall [83] also makes use of model checking for lookahead analysis and execution steering. Here, nodes in a distributed system run continuously a state exploration algorithm on a recent snapshot of their neighbourhood with the aim of detecting future inconsistencies. When such inconsistencies are detercted, the ongoing execution can be steered away from the predicted errors. However, CrystalBall addresses well-specified distributed systems. Also, while CrystalBall uses execution steering in a defensive way, to protect itself from errors, our method employs similar techniques aggressively, for runtime exploration of relevant black box component behaviour.

An interesting white-box technique, that relies on a combination of symbolic and concrete execution, and model inference is [25], by Cho et al. The MACE approach builts an abstract model of the program as a finite state machine, and uses this model further on to guide space state exploration. As new behaviour is explored, the model is refined. This method is related to ours in that it combines model inference and dynamic state space exploration, however it specifically addresses white-box components.

Chapter 4

Learning Safe Behavioural Models for System Composition

4.1 Overview

Let us start from the assumption that we need to compose a system out of a set of components, and that the system should comply to a certain specification. For at least some of the needed components no behavioural model has been provided.

The interactions in the system to be built are asynchronous, taking place as sequences of message send/receive events, as it is the case in asynchronous event-based architectures. By asynchronous interaction we understand the specific type of communication in which an entity can send a message and then continue its execution, independently whether the message was received or not, as opposed to synchronous interaction, where the execution of the sender is blocked until the sent message is received. An additional assumption we make is that, when the destination entity receives the sent message, it sends an acknowledge message to the sender. Asynchronous communication is important, especially in event-driven systems, as it prevents the component from uselessly blocking in its execution.

Communication in the system happens by means of message queues or buffers, of bounded capacity, that connect the entities in the system, i.e. the components, among themselves. The messages do not overtake each other.

It is assumed that the messages sent or received by a component belong to a finite set of message types, which defines the event alphabet of the component. Both send and receive events are assumed observable. From any state, a component can have maximum one outgoing transition on a specific event, no matter if that event is a message send or a message receive event. Thus, an asynchronous component can be modelled as a finite state machine that is both input deterministic, and output deterministic. However, a component can have more than one send transition from a state, and the choice between sending one message or another is taken internally, therefore the finite state machine modelling the component's behaviour is not output determined, and, thus, it's also not behavioural deterministic.

Each component can be reset anytime to its initial state.

We have already assumed that the system to be built must comply to a provided specification, or composition goal. This composition goal is represented as a temporal safety property given as a finite state machine over a set of send/receive events. It describes all the interaction sequences to be allowed in the system. Compliance is considered as language inclusion: the language of the system must be included in the language of the specification.

To achieve the composition goal, i.e. the desired system conforming to it, we must find a way to enforce this property on the set of interconnected, interacting components. For this, the straightforward solution is to build an adaptor, i.e. a component in the middle that properly coordinates the interactions in the system towards satisfying the desired property. Further on, we will think of this adaptor as a controller over the system plant, which restricts possible executions to allowed interaction sequences.

In classic control theory [67], with respect to the alphabet of a component, a distinction is made between controllable events, which can be enabled or disabled from outside, and uncontrollable events, which cannot be forced from the exterior and occur as nondeterministic outputs. The problem of synthesizing a controller that enforces a certain property over a plant is solved by using the controllable events of the plant to disable any event sequences that might violate the specification. In our case, the controllable set of events is considered to be the set of message receive events, since it can be decided from outside what message to forward to the component, and when, while the set of message send events is considered uncontrollable, since message sends are based on internal decisions of the component and more than one message send event may occur from the same component state.

Normally, in order to build a controller over the system plant, formal behavioural models, describing both controllable and uncontrollable behaviour, are needed for all components. In the case of a plant consisting of a set of black box components, controllable and uncontrollable behaviour cannot be precisely known. However, it is important to observe that, if a controller can be synthesized using an underapproximation of the real controllable behaviour and an overapproximation of the real uncontrollable behaviour of the plant, the plant can still be controlled, although the resulting controller is likely to be more restrictive than one built for a precise model.

We call such a model, which underapproximates controllable and overapproximates uncontrollable behaviour of the real black box, a safe approximation. The underapproximation and overapproximation are defined as follows: if a controllable event appears after a prefix in a sequence accepted by the approximate model, it will also appear, after the same prefix, in the precise model; respectively, if an uncontrollable event appears after a prefix in the real model, it will appear after the same prefix in the approximate model. Thus, the language of the approximate model is controllable with respect to the real, unknown model (as defined in section 2.5).

As at least some of the components to be used are black boxes, synthesizing a controller for the system requires that their behavioural models are learned first. As black box model learning can only use information obtained from runtime observation, and as runtime exploration can be costly, in most cases only a subset of the actual behaviour will have been explored. Thus, the inferred model will actually approximate, to some degree, the actual behaviour of the real component.

However, this approximation must satisfy certain requirements. More specifically, it is a safe approximation of the real behaviour that has to be inferred for all the black box components in the system, so that an effective controller can be obtained. Therefore, the actual problem is that, from a bounded number of executions, a model has to be built that would underapproximate only the controllable behaviour and overapproximate its uncontrollable one.

For such a model to be as useful as possible with respect to the mentioned control problem, two conditions are important. First, the explored executions should cover as many sequences of events as possible. Second, as there is no point in learning behaviour that is actually prohibited by the specification, the event sequences not allowed by the desired property are filtered out during learning. Thus, the models learned are not only approximate, but also partial models of the

real black box components, since only behaviour accepted by the specification is covered. Therefore, we must generate executions that respect the temporal specification, while covering as many event sequences as possible.

We aim to infer these models as finite state machines. To infer models for the black box components in the system we need to determine and observe execution sequences that explore the controllability of correct system behaviour. We say we enable a controllable transition whenever we forward to the component the message it needs to receive for the transition to be triggered.

To explore only certain execution sequences, the plant has to be controlled at runtime, in order for each run to be steered towards paths accepted by the specification. Under this purpose, we place an intelligent, proactive adaptor among the components, that will intercept all sent messages and control the system by forwarding or consuming them, while monitoring the components to see if they did accept earlier forwarded messages. In the same time, observed events are used to synchronize the current execution with a path in the specification automaton.

The information on explored sequences, both failed and successful, is kept in a tree structure for each black box component in the system. Each node corresponds to an observed prefix in the unknown language of the black box component, and it contains links towards the next observed events. Each such link leads towards a suffix sub-tree. There are three types of links: valid links, corresponding to actually observed events, empty links, that do not lead towards any suffix sub-tree, their purpose is to mark those events that failed to be enabled after a certain prefix, and unexplored links, corresponding to yet unexplored events.

The behaviour exploration takes place depth-first. We assume a maximum bound m on the depth of the black box behavioural model, thus, any acyclic path in the model is assumed to contain at most m transitions. The depth-first exploration works with sequences of up to $2m$ events, in order to properly identify the cycles in the model.

We conduct the runtime experiments in the following way. To start a new execution, all components in the system are reset, the sent messages are intercepted by the proactive adaptor, and the specification automaton is advanced according to the observed event trace. Then, suppose that, for a black box component, the current prefix corresponds in its trace tree to a node that has more than one unexplored link on message receive events, and that more than one of these

messages are available, i.e. have already been sent by one of the other components in the system. So, a set of yet unexplored controllable events exists for the mentioned component, following the current prefix, and these events can be enabled by the proactive adaptor.

The specification automaton describes a safety property. If, following the current event sequence, a transition exists in the specification automaton on one of the previously mentioned unexplored controllable events, then that transition will be explored at runtime, as it conforms to the safety property. If several such transitions exist, they will be explored in a random order. If the message is forwarded to the component, but the component does not accept it, this is marked by an empty link in corresponding the node from the component tree, and another controllable event is enabled. Otherwise, the exploration continues from the new prefix.

It is important to observe that, while through model checking all outgoing paths can be simultaneously explored, a controlled execution can only explore one path, and needs to be restarted for further exploration. Due to the existence of uncontrollable events, returning to a previously observed prefix might be difficult, requiring several attempts, thus a maximum of information should be obtained out of each run. This is why we have favoured depth-first over breadth-first exploration.

Whenever an execution violates the specification, or the maximum execution trace length of $2m$ events is reached, the ongoing run is forced to end by resetting all components in the system.

One detail omitted so far is that every node, in each tree structure associated to each black box component, has a counter, that keeps track of the number of times the node is reached, i.e. its corresponding prefix has been observed. The components in the system are considered to have the following property, known as bounded fairness [32]: if a component has reached a state for more than θ times, where θ is the fairness bound, then every uncontrollable transition from that state has been triggered at least once. Since the component is deterministic, an observed sequence over the component alphabet cannot lead to more than one state in a hypothetical real model of the component behaviour. Therefore, for each node in the tree structure, when the counter associated to the node reaches θ, the message send events that haven't been observed can be considered not to occur after that prefix, and their corresponding links become empty links.

A node in a tree associated to a black box component is considered completely

explored when it has no missing links. The exploration process stops when, for all black boxes in the system, in their corresponding trees, all the nodes at a depth less than or equal to $m + 1$ are completely explored. It also ends when a previously established maximum number of executions is reached, if the former termination condition hasn't been met up to that point.

If the maximum number of executions was reached and there still are incompletely explored nodes at depths smaller than $m + 1$, then all links from these nodes, that correspond to unexplored message receive events, become empty links. This is done since controllable behaviour can be underapproximated by pruning out unobserved sequences.

After this last step, the model for the black box component is built from the trace tree in the following way. First, each node becomes a state, and each valid link a transition. Then, two completely explored states are considered equivalent if they simulate each other on a depth of m, i.e. a sequence observed from one state is either observed, or not tried from the other. Also, two states are considered equivalent if their "parent" states are equivalent, and their incoming transition is labeled by the same event – as they were initially nodes in a tree, each state has only one incoming transition.

Based on this equivalence relation, each inferred model is minimized. We then compose the system by building the adaptor as the controller over the system plant.

4.2 Assumptions

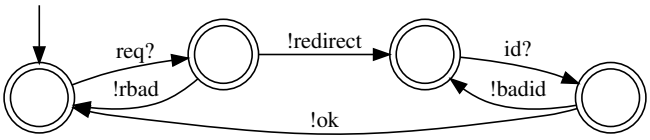
We consider a system S to be composed out of a set $\mathcal{C}$ of asynchronous deterministic components, out of which n are black boxes. The asynchronous components communicate by means of bounded buffers, i.e. input and output ordered message queues.

4.2.1 Component Models

Each component C_i is associated to a model M_i, which describes its behaviour. This model is described by a finite state machine, defined as a 5-tuple

$$M_i = \langle Q^i, q_0^i, Q_f^i, \Sigma^i, \delta^i \rangle$$

- Q^i is the state set
- $q_0^i \in Q^i$ is the initial state
- Σ^i is the event set of C_i
- $\Sigma = \bigcup \Sigma^i$ is the event set of the system
- $Q_f^i = Q^i$ is the set of accepting states
- $\delta^i : Q^i \times \Sigma^i \rightarrow Q^i$ is the partial transition function



(a)

Figure 4.1: (a) Finite state machine with message send/receive events.

The component behaviour is modelled using a deterministic finite state machine in the sense defined at section 2.3, i.e for every state and event there is at least one successor state. Also, since all its states are accepting, $Q_f^i = Q^i$, the language defined by the finite state machine is a prefix closed language. Taking into account the types of automata described at section 2.3, this behavioural model can also be seen as a Mealy machine that can have both empty inputs and outputs, and can nondeterministically accept empty inputs. It can also be regarded as an input and output deterministic, but not output determined interface automaton.

An event σ in any of the component event sets Σ^i is either a message send: $msg!$, or a receive: $msg?$. Let us denote by $\exists\delta^i(q,\sigma)$ the fact that function δ^i is defined in state q, for event σ. We denote by $\Sigma^i(q)$ the set of events σ for which $\exists\delta^i(q,\sigma)$, with its two subsets $\Sigma^i_?(q)$ – the subset of controllable events, and $\Sigma^i_!(q)$ – the subset of uncontrollable events.

Considering a string of events $t_k = \sigma_0\sigma_1..\sigma_k$, and Σ^i_* the set of strings formed from events in Σ^i, then the extension δ^i_* of the partial transition function δ^i to strings of events is the string transition function δ^i_*, also a partial transition function, which is described below.

$$\delta_*^i : \Sigma_*^i \to Q^i$$

$$\delta_*^i(t_k): \quad \delta_*^i(t_0) = \delta^i(q_0, \sigma_0) \text{ and}$$

$$\delta_*^i(t_k) = \delta^i(\delta_*^i(t_{k-1}), \sigma_k)$$

If a trace $t \in \Sigma_*^i$ is observed at runtime, we denote this fact by $obs(t)$. If an event $\sigma \in \Sigma^i$ is observed from a state $q \in Q^i$, we denote this by $obs'(\sigma, q)$.

We assume a bounded fairness [32], considering a threshold θ on the number of state visits for a single state q of the model as a fairness bound. This means that if a state q is reached for at least θ times, every uncontrollable event σ enabled in q is observed at least once:

$$\sigma \in \Sigma_!^i(q) \wedge visits(q) \geq \theta \longrightarrow obs'(\sigma, q)$$

The components communicate in a peer-to-peer way, by means of bounded buffers, which represent dedicated message channels. These are ordered messages queues, using a first-in-first-out (FIFO) processing technique. Thus, the first message sent by a component will also be the first received by its peer.

A bounded message buffer of bound b that connects the output of a component C_i to the input of a component C_j can be described by a finite state machine B_{ij}. Its alphabet Σ^B will contain those send events in Σ^i that have a mirror, i.e. a corresponding receive event in Σ^j, and these mirror receive events from Σ^j. Thus, $\Sigma^B = \Sigma_!^B \cup \Sigma_?^B$, where

$$\Sigma_!^B = \{\sigma \in \Sigma_!^i \mid \sigma = msg! \to \exists \sigma' \in \Sigma_?^j \; s.t. \; \sigma' = msg?\}$$

$$\Sigma_?^B = \{\sigma \in \Sigma_?^j \mid \sigma = msg? \to \exists \sigma' \in \Sigma_!^i \; s.t. \; \sigma' = msg!\}$$

Each state q in Q^B is accepting and is characterized by a number of $|\Sigma_!^B|$ counters, and each counter k_σ is associated to an event $\sigma \in \Sigma_!^B$, thus we have $q(k_{\sigma_0}, k_{\sigma_1}, ...)$. For each state q, the sum of all its counters k_σ is at least 0 and at most b, where b is the size of the buffer: $\sum k_\sigma \leq b$. The initial state q_0 has all counters k_σ on zero, which is denoted as $q_0(0, 0, .., 0)$.

Each transition on an event σ' in $\Sigma_!^B$ from a state q leads towards a state q' which has all counters identical to the counters of q, except for $k_{\sigma'}$, which increases by 1. A transition on a message receive event $\sigma' \in \Sigma_?^B$ from a state q only exists if the counter corresponding to σ'', the mirroring of σ' has a value greater than

0, and leads towards a state q' which has all counters identical to the counters of q, except for $k_{\sigma''}$, which increases by 1.

Then, the partial transition function δ^B of B_{ij} is defined as following:

- $\delta^B(q(..., k_{\sigma'}, ...), \sigma') = q'(..., k_{\sigma'} + 1, ...)$
  for q, q' with $0 \leq \sum k_\sigma < b$ and $\sigma' \in \Sigma^B_!$
- $\delta^B(q(..., k_{\sigma''}, ...), \sigma') = q'(..., k_{\sigma''} - 1, ...)$
  for q, q' with $0 < \sum k_\sigma \leq b$, $\sigma' \in \Sigma^B_?$, where σ'' mirrors σ' and $k_{\sigma''} > 0$

Similar bounded buffers are used between a component C_i and any special component-in-the-middle that has the purpose of coordinating the interactions in the system, such as a proactive adaptor (used for learning), or an adaptor (a controller component that uses message forwarding and consuming to enable, respectively disable controllable events in the system). We assume that when such components are present, they intermediate all interactions in the system, and no pair of ordinary components is directly connected by message queues.

The desired system S must comply with a property Φ, also described by a finite state machine.

$$\Phi = \langle Q^\Phi, q_0^\Phi, Q_f^\Phi, \Sigma^\Phi, \delta^\Phi \rangle$$

- Q^Φ is the state set of the property automaton
- $q_0^\Phi \in Q^\Phi$ is the initial state
- $\Sigma^\Phi = \Sigma$ is the event set of the property, including all the events in the system
- $Q_f^\Phi \subseteq Q^\Phi$ is the set of accepting states
- $\delta^\Phi : Q^\Phi \times \Sigma \to Q^\Phi$ is the partial transition function

Φ is a safety property, describing the allowed sequences of events in the system. One should note that, while the automata that specify component behaviour have all their states as accepting, as they describe prefix-closed languages, things are different for Φ. In the case of property automaton Φ, accepting states Q_f^Φ may represent only a subset of the state set $Q_f^i \subseteq Q^i$, thus the language of Φ is not necessarily prefix-closed. The event set of the system includes the event set of the property automaton Φ: $\Sigma \supseteq \Sigma^\Phi$. The compliance relation considered is language inclusion: all the sequences of events allowed in the system S must be included in the language of Φ: $\mathcal{S} \subseteq \mathcal{L}(\Phi)$.

Let us also assume that two special, auxiliary events $ack!$ and $rst?$ exist, such that $\forall i \leq n-1.\ ack!, rst? \in \Sigma^i$. Event $ack!$ confirms a successful receive: it is emitted by component C_i when a message msg arrives in a state q of C_i in which it can be accepted: $\delta^i(q, msg?) \neq \emptyset$. Although event $ack!$ is emitted right after a message was accepted, its destination component receives it within a maximum of μ microseconds, where μ is the bounded acknowledgement delay. However, an asynchronous component does not wait for any acknowledgment from its peers and its execution progresses independently of it. Event $rst?$ forces any of the components to return to its initial state, from any state q. These assumptions are important for our learning algorithm, since event $ack!$ makes receive events externally observable and event $rst?$ allows reset.

Suppose that we try to enable an event σ in component C_i, and C_i is in a state q from which no outgoing transition on σ exists, so it does not accept the forwarded message. Then, the forwarded message is ignored, no transition is taken, and thus, component C_i does not change its state.

Furthermore, in order to ensure termination, we consider a bound m on the maximum acyclic path in the model, for any component C_i. Usually, in model learning, it is the maximum number of states that is bounded, however, while in some extreme cases the maximum acyclic path equals the number of states in the model, minus one, usually it is much smaller. Also, our algorithm needs an upper bound for the size of the sample traces used in learning, but, unlike most learning techniques, it does not need to know the maximum state size of the target automaton.

In order to differentiate an unknown, learned model of a component from M_i, the model of a known component, let us note the learned model of a black box component with U_i. The learned model U_i will be an approximation of the real behaviour of the component C_i, which we assume can be precisely described by an unknown finite state machine R_i.

4.2.2 Trace Trees

During the learning process, the event traces observed to belong or not to the language of a component C_i are kept in a trace tree, which is continuously updated as runtime observations are made. Therefore, we will have a trace tree T_i for every black box component C_i in the system.

A trace tree T_i is a structure modelled by a quadruple:

$$\langle V^i, v_0^i, \Sigma^i, \xi^i \rangle$$

- V^i is the set of all vertices in the tree. It includes also the null vertex $\{\epsilon\}$, which is used to mark a dead end in a trace (when, for example, a controllable event could not be enabled, etc.)
- v_0^i is the initial vertex, the root of tree T_i
- Σ^i is the set of events associated to tree T_i, and it is the same as the set of events of C_i
- $\xi^i : V^i \times \Sigma^i \to V^i$ is the partial transition function

The transition function ξ^i defines the edges between vertices in the tree T_i. For every event $\sigma \in \Sigma^i$, if the transition function is defined for a vertex $v \in V^i \setminus \{\epsilon\}$, so that $\exists \xi^i(v, \sigma)$ and $\xi^i(v, \sigma) = v'$, then no other vertex v'' exists so that $\xi^i(v'', \sigma) = v'$. This encodes the essential tree property: each node has a unique parent.

$$\forall \sigma \in \Sigma^i. \quad \forall v, v', v'' \in V^i \setminus \{\epsilon\}. \quad (\xi^i(v, \sigma) = v' \wedge \xi^i(v'', \sigma) = v') \to v = v''$$

For the null vertex ϵ, the transition function ξ^i is undefined for all events in the event set: $\forall \sigma \in \Sigma^i.\ \nexists \xi^i(\epsilon, \sigma)$.

Similarly to the case of the finite state machines used, we define the string transition function ξ_*^i as the extension to strings of events of the transition function ξ^i. We consider a string of events $t_k = \sigma_0 \sigma_1 .. \sigma_k$.

$$\xi_*^i : \Sigma_*^i \to V^i$$

$$\xi_*^i(t_k): \quad \xi_*^i(v_0^i) = \xi^i(v_0^i, \sigma_0) \text{ and}$$

$$\xi_*^i(t_k) = \xi^i(\xi_*^i(t_{k-1}), \sigma_k)$$

One important property of the trace tree is that the restriction of the string transition function ξ_*^i to $V^i \setminus \{\epsilon\}$ is an injection. Thus for every vertex $v \in V^i \setminus \{\epsilon\}$ there is only one path in the tree from the root vertex v_0^i to v, which contains no cycles, since any tree is a directed acyclic graph.

$$\forall v \in V^i \setminus \{\epsilon\}. \quad \forall t, t' \in \Sigma_*^i. \quad (\xi_*^i(t) = v \wedge \xi_*^i(t') = v) \rightarrow t = t'$$

The trace t for which $\xi_*^i(t) = v$ is called the access trace of vertex v, where v is different from the null vertex, $v \neq \epsilon$.

Consider a sequence of events $t \in \Sigma_*^i$. We know that the trace tree T_i contains the results of all runtime observations made on the behaviour of component C_i. Then, if the string transition function ξ_*^i is defined for t, i.e. $\exists \xi_*^i(t)$, we consider that the sequence of events t was successfully enforced at runtime on black box component C_i, which we denote by $positive(t)$. If trace $t'' \in \Sigma_*^i$ is a prefix of the observed trace t, then t'' is also observed: $positive(t) \rightarrow positive(t')$.

If for the trace $t \in \Sigma_*^i$ a prefix t' of t exists, so that $\xi_*^i(t') = \epsilon$, this means that trace t' was tried at runtime on C_i, and the attempt failed, so t' does not belong to the language of the black box C_i. This is denoted by $negative(t')$. Since t' is a prefix of t, this means $negative(t') \rightarrow negative(t)$.

If no prefix t' so that $negative(t')$ exists for trace $t \in \Sigma_*^i$, but the string transition function ξ_*^i is undefined for t, then it means that string t has not been tried yet on C_i, which is denoted by $untried(t)$. So, a sequence that hasn't been marked neither as a negative, nor as a positive sample, is yet to be experimented with: $\neg positive(t) \wedge \neg negative(t) \rightarrow untried(t)$. Also, if untried trace t is a prefix of another trace $t'' \in \Sigma_*^i$, then t'' is also untried: $untried(t) \rightarrow untried(t'')$

Thus, the trace tree is used in retaining positive and negative samples obtained during learning, while also keeping track of the paths yet to be explored. As the model U_i to be learned exhibits a bounded fairness, it is also useful to keep track of the number of times a certain prefix t' is observed. Therefore, for each vertex $v \in V^i \setminus \{\epsilon\}$ a counter function is defined $\nu : v \in V^i \setminus \{\epsilon\} \rightarrow \mathbb{N}$, which associates the vertex v with the number of times $\nu(v)$ its access trace is observed at runtime, during the learning process.

Let the depth of a trace tree T_i be the length of a maximum trace $t \in \Sigma_*^i$, so that the string transition function is defined for t, i.e. $\exists \xi_*^i(t)$. The learning experiments will be performed using traces in Σ_*^i of a length bounded to $2m$, where m is the maximum acyclic trace length assumed for component C_i. The execution traces are bound to $2m$ because, first, up to depth m we can still

discover new states, and, second, we need to know all possible behaviour from each state on a depth of m in order to determine state equivalence.

Thus, any trace tree T_i associated to a black box component C_i will have a maximum depth of $2m$.

4.3 Centralized Behaviour Exploration

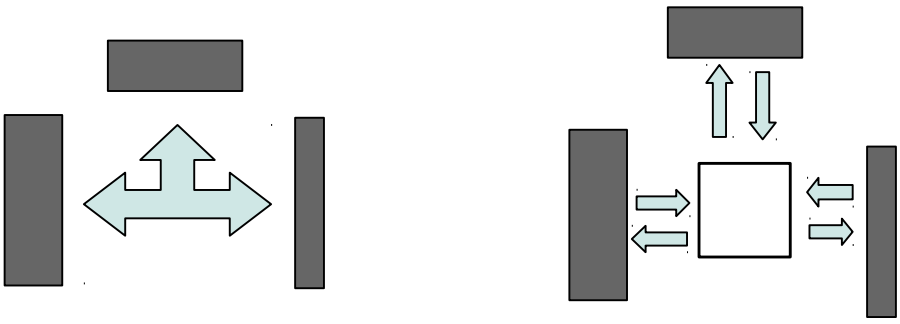
For simplicity, and without loss of generality, we will further on consider the case where all components C_i in the system are incompletely specified, i.e. black boxes, and $|\mathcal{C}| = n$.

The main characteristic of the centralized behaviour exploration process is that all the components in the system are executed and controlled together, and the execution traces enforced are only those that conform to the desired property Φ. This happens independently of the number of black box components that are in the system.

Enforcing event sequences that conform to the safety property Φ has the advantage of focusing the learning effort on the specific behaviour required in the system to be composed. Thus, for a black box component C_i, the learned model U_i will describe in many cases just part of the real behaviour, hypothetically modeled by R_i. Traces in $R_i \| \overline{\Phi}$ will not be tried on C_i. Saving part of the behaviour exploration effort is important, since the actual number of runtime queries used is usually the most expensive part of model learning.

For model learning under a composition purpose it is important that the behaviour described by the learned U_i is useful for building the system. Most real applications interact with their user in a loop, and not having to reset a component or a system to its initial state too often is a valuable feature. Since we aim to model a safe approximation of the real black box behaviour, an important point is correctly identifying the cycles in the model. The important cycles in a component's behaviour are the ones that can also be found in the property automaton. Since the learning process can be limited in time, or in the allowed number of executions, the exploration process should focus on a faster identification of the cycles in the property automaton in order to obtain a more useful inferred model.

In figure 4.2 we can see the setup for the learning process. A proactive adaptor is placed among the components in the system. Its purpose is to control all the



(a) (b)

Figure 4.2: Learning Setup: 4.2(a) before: components interact unrestricted, 4.2(b) after: all interactions take place through the proactive adaptor.

interactions in the system, by intercepting all messages sent by the components and by forwarding or consuming them in order to enable or, respectively, disable message receive events in components. Thus, the proactive adaptor acts like a system controller (see section 2.5), only that its decisions of forwarding/consuming messages or resetting the system are taken dynamically, based on runtime observations. Also, the purpose of the proactive adaptor is to continuously explore new correct paths in the system, as opposed to only enforcing correct behaviour, as it would be the case for a real controller. So, by properly coordinating the interactions in the system, new samples, both positive and negative, are obtained for the black box components to be learned.

It is worth noting here that during the learning process, when the behaviour of the components in the system is explored at runtime, the message buffers that connect the proactive adaptor with the components contain a maximum of one message at a time, as the proactive adaptor waits to see whether the message it sent was accepted or not by the destination component. Also, a message sent by a component is consumed by the proactive adaptor as soon as it arrives. Therefore, the actual size of the buffers is less important for the learning process.

Controlling a Current Execution

First, lets see how we would try to enforce correct sequences of events at runtime, if we had precise behavioural models for all the n components in the system: M_0, M_1, ..., M_{n-1}.

Let us denote by $M_\times$ the asynchronous product of the models $M_\times = M_0 \times M_1 \times ... \times M_{n-1}$, as defined in section 2.3. If q is a global state, we denote its projections on M_i as q^i. We then consider a control point in $M_\times$ as a global state q_{cp} for which at least one of the outgoing controllable event sets $\Sigma^i_?(q^i_{cp})$ contains 2 or more receive events: $|\Sigma^i_?(q^i_{cp})| \geq 2$. This basically means that in model M_i, from local state q^i_{cp} we can choose from at least two possible execution paths. We call such a local state q^i_{cp} an active projection of the control point q_{cp}.

We say that an event σ is observed in our learning setting when either the proactive adaptor intercepts a sent message, or when the proactive adaptor forwards a message to a component and the message is successfully received.

During the exploration process, the following happens: whenever an event σ, where $\sigma \in \Sigma$ and $\sigma \in \Sigma^\Phi$ is observed, a transition on σ from the current state is taken both in the product machine $M_\times$ and in the property automaton Φ. The execution is thus matched with both the exploration of the product model and of the specification. The two automata are both advanced by the same observed event σ, concomitantly with the ongoing execution. If the product $M_\times$ is too large, we can avoid to actually compute it: assuming that $\sigma \in \Sigma^i$, then a transition on σ from the current state is taken in M_i and the individual component model is the one advanced together with Φ.

When the execution reaches a control point q_{cp} in $M_\times$, with a local active projection q^i_{cp}, a receive event $\sigma \in \Sigma^i_?(q^i_{cp})$ is enabled for execution if from the current state of Φ a transition triggered by σ exists. Thus, its corresponding message is forwarded to component C_i, so that the message receive event can be triggered in the current run.

If, from a state q, an uncontrollable event σ' occurs at runtime, for which no transition triggered by σ' exists in Φ from the current state, then we observe that property Φ is violated at runtime, from state q in an uncontrollable way.

The execution is forced to end when either the property Φ has been violated, or the current execution trace has reached a maximum number of steps.

Black Box Exploration

Now, since no models M_i are available for the n components in the system, the only reference we have to coordinate the learning executions is the property automaton Φ. We use the trace trees T_i to establish priorities between different sequences to be tried on, to store both positive and negative samples and to define a termination condition for the learning process.

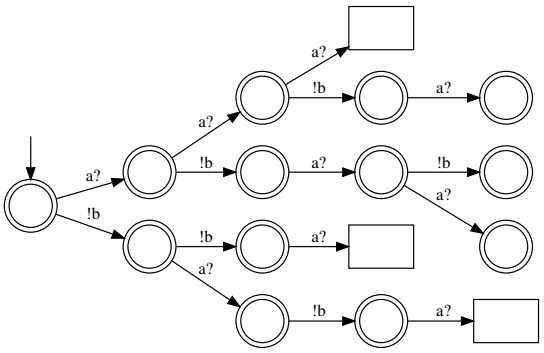
As mentioned in the previous section, each black box component C_i is associated to a trace tree T_i. Let us assume the current execution trace is t, and that is has led the property automaton Φ to a state q. Let $t_{|i}$ denote the projection of trace t on the event set Σ^i, corresponding to component C_i.

Let us assume an event $\sigma \in \Sigma^i$ occurs. In property automaton Φ, the transition on Φ from the current state exists and is taken. Then, the current trace becomes $t\sigma$, and, if the projection $t_{|i}\sigma$ is a new trace, it has to be marked as observed in the tree T_i. Let $v = \xi^i_*(t_{|i})$ be the vertex returned by the string transition function of T_i. Since $t_{|i}\sigma$ hasn't been observed before, $\xi^i(v, \sigma)$ is still undefined. Therefore, a new vertex v' is added to V^i, such that the transition function ξ^i changes to $\xi^i(v, \sigma) = v'$, while $\xi^i(v', \sigma')$ is undefined for all $\sigma' \in \Sigma^i$. The counter function for v' becomes $\nu(v') = 1$, while remaining unchanged in rest. Also, the remainder of the transition function ξ^i stays the same.

If, however, $t_{|i}\sigma$ is a trace already in T_i, and vertex v' so that $v' = \xi^i_*(t_{|i}\sigma)$ already exists, then the only thing that changes is the counter function, since trace $t_{|i}\sigma$ has been observed once more. Therefore, the result returned by the counter function for vertex v' increases with 1: $\nu_{new}(v') = \nu_{old}(v') + 1$.

So far, we have only seen how to react to an occurring event. Let us see now how the actual exploration takes place. As before, let us assume the current trace is t, and that property Φ is in a current state q. Suppose q is a control point, with an active local projection q^i – if q has several such active projections, one of them is randomly chosen. Suppose also that we have to choose between two controllable events, σ and σ', which are both in the event set Σ^i of component C_i. We now have to decide which event to enable in C_i, between the two. We have three possible cases:

- none of the traces $t_{|i}\sigma$ and $t_{|i}\sigma'$ has been previously tried: $untried(t_{|i}\sigma) \wedge untried(t_{|i}\sigma')$, so we randomly choose which event to enable
- only $t_{|i}\sigma$) has been previously tried: $(positive(t_{|i}\sigma) \vee negative(t_{|i}\sigma)) \wedge untried(t_{|i}\sigma')$, so it is event σ', the untried one, which is enabled



(a)

Figure 4.3: The trace tree during the exploration process

- both $t_{|i}\sigma$ and $t_{|i}\sigma'$ have already been observed: $positive(t_{|i}\sigma) \wedge positive(t_{|i}\sigma')$; we want to explore the behaviour they may lead to in an uniform way, so we will choose the least beaten path: considering vertices $v = \xi^i_*(t_{|i}\sigma)$ and $v' = \xi^i_*(t_{|i}\sigma')$, we will enable event σ if $\nu(v) \leq \nu(v')$ and σ' otherwise.

Let us now suppose we find ourselves in the first case, we try to enable event σ, and component C_i does not accept the forwarded message. In this case, as assumed, no transition has been taken and C_i hasn't changed its state. Therefore, we can continue from that point on by trying σ', or other controllable events, if available. As trace $t_{|i}\sigma$ could not be enforced on C_i, this will be marked in T_i by modifying the transition function from vertex $v = \xi^i_*(t_{|i})$, so that, for event σ, it leads towards the empty vertex: $\xi^i(v, \sigma) = \epsilon$. A trace tree with several observed, several unexplored and several explored, but found infeasible traces can be seen in figure 4.3, where the square-shaped nodes mark the empty vertices in the tree, and the parameter m is set to 2.

A vertex v in a trace tree T_i is considered completely explored, which is denoted by $complete(v)$, if:

- the transition function ξ^i is defined for all the controllable events $\sigma \in \Sigma^i$: $\forall \sigma \in \Sigma^i_?.\exists v' \in V^i$. s.t. $\xi^i(v, \sigma) = v'$
- the access trace t of vertex v has been observed for at least θ times, when θ is the fairness bound: $\nu(v) \geq \theta$.

A trace tree T_i is considered completely explored to depth p, which is denoted as $complete(T_i, p)$, if all traces $t \in \Sigma^i_*$ of length smaller or equal to p, are either stored or marked as infeasible in T_i, and all stored traces are access traces to

completely explored vertices $v \in V^i \setminus \{\epsilon\}$:

$$\forall t \in \Sigma_*^i \; s.t. \; |t| \leq p \wedge \xi_*^i(t) = v \wedge complete(v) \rightarrow complete(T_i, p)$$

The behaviour of a black box component C_i is considered completely explored when its associated trace tree T_i is complete to depth $m+1$: $complete(T_i, m+1)$.

A current execution is forced to end when either the property Φ has been violated at runtime, or when each projection of the current execution trace on the alphabet of a black box component has reached the maximum number of steps $2m$: $\forall i < n. \; |t_{|i}| \geq 2m$.

The centralized behaviour exploration process ends when either a maximum allowed number γ of executions has been reached, or all the black box components in the system have been completely explored.

4.3.1 Centralized Exploration Algorithm

Below we give a pseudocode description of the behaviour exploration algorithm for safety properties. The algorithm explores the behaviour globally, with respect to all the components in the system.

Algorithm 1 : Behaviour exploration controllable step

1: **procedure** ctrlStep {control an execution step}
2: **for all** $i = \overline{0, n-1}$ **do**
3: $progress = false$
4: $v = nodeAfterTrace(T_i, t_i)$ {current vertex in T_i}
5: $toTry = availableMoves(v, i) \cap correctMoves(q^{\Phi}, \Phi)$
6: **repeat**
7: $leastTaken = \{\sigma' \in toTry \mid minim(\nu(\xi^i_*(t_i\sigma')))\}$
8: $\sigma = pickRandom(leastTaken)$
9: $enable(\sigma, C_i)$ {try message}
10: **if** ack_i **then**
11: $advance(\Phi, \sigma)$ {advance property}
12: $putTrace(T_i, t_i\sigma)$ {positive sample}
13: $t_i = t_i\sigma$ {augment current trace}
14: $count_i = count_i + 1$ {count step}
15: $progress = true$ {execution progress}
16: **else**
17: $cutTrace(T_i, t_i\sigma)$ {negative sample}
18: **until** $progress$ {step ends if execution progress}
19:
20: **procedure** onReceive(σ) {message receive listener}
21: $i = \sigma.source$
22: $msg = \sigma.message$
23: $store(msg)$ {store received message}
24: $putTrace(T_i, t_i\sigma)$ {positive sample}
25: $t_i = t_i\sigma$ {augment current trace}
26: **if** $advance(\Phi, \sigma)$ **then**
27: $count_i = count_i + 1$ {count step}
28: $progress = true$ {execution progress}
29: **else**
30: $stopCurrentExecution = true$ {violated property}

Algorithm 2 : Exploring behaviour at runtime

1: {termination condition}
2: def completelyExplored$(T_i) = \forall v \in V^i.\ depth(v) \leq m+1 \rightarrow \nu(q) \geq \theta$
3: {execution end condition}
4: def maxTrace $= (\forall i = \overline{0, n-1}.\ count_i \geq maxExLength)$
5:
6: **procedure** execution($maxExLength$) {control an execution}
7: $stopCurrentExecution = false$
8: **for all** $i = \overline{0, n-1}$ **do**
9: reset C_i
10: $count_i = 0$
11: **repeat**
12: $progress = false$
13: ctrlStep
14: **until** $\neg progress \ \vee stopCurrentExecution \ \vee \ maxTrace$
15:
16: {behaviour exploration main}
17: **for all** $i = \overline{0, n-1}$ **do**
18: initialize(T_i)
19: $execNr = 0$
20: $maxExLength = 2m$
21: **repeat**
22: execution($maxExLength$)
23: $execNr = execNr + 1$
24: **until** $(execNr \geq maxExecNr) \vee (\forall i \leq n-1.$ completelyExplored$(T_i))$

Procedure **ctrlStep** describes one step of the exploration process. For each component i, the set $toTry$ of controllable events is determined as the intersection of the set of available moves with the set of correct ones. First, function $correctMoves$ returns all controllable events that can be enabled from the current state q^{Φ} in the property model. Out of these receive events, we only try to enable the ones for which we have an available message to forward, i.e. the needed message has already been received and stored by the proactive adaptor. This event set is returned by function $availableMoves$. Then, a controllable event σ to be enabled is chosen. Each time, the chosen event σ is the one that has been enabled the fewest times so far (possibly never), after trace t_i, from all controllable events available. If several such controllable events σ exist, the enabled event is picked randomly from this subset.

If the enabled message receive event σ has been acknowledged, operation $advance$ synchronously advances the property model on a transition triggered by σ. Event σ is concatenated to the current execution trace, the number of execution steps performed by component C_i is incremented and progress is marked. The observed trace $t_i\sigma$ is kept as a positive sample in the trace tree T_i, by means of operation $putTrace$. Otherwise, trace $t_i\sigma$ is marked as infeasible in the trace tree T_i, by means of operation $cutTrace$.

Procedure **onReceive** is a handler procedure, listening for asynchronously incoming messages. It is performed each time a message sent by a component is intercepted by the proactive adaptor. First, the sent message is stored by the proactive adaptor for possible later use – by operation $store$. Then, uncontrollable event σ is appended to trace t_i, $putTrace$ adds observed trace $t_i\sigma$ as a positive sample in the trace tree T_i. If the property model can accept the observed send event σ, the number of execution steps performed by component C_i is incremented and progress is marked. If not, then we have reached an uncontrollable erroneous transition, and therefore the current execution will end. There is no need to further explore this path, if a controller will be successfully generated for the system, then it will disable all paths that violate Φ.

The course of one exploratory execution is described in procedure **execution**. In the beginning, all components are reset, and each execution step counter is set to 0. Then, for each step to be taken in the execution, the progress variable is set to false, after which a controllable exploration step is taken through procedure $ctrlStep$. Incoming messages are received in an asynchronous manner, handled and stored. The execution stops when either a number of $2m$ steps has been taken by each black box component in part, or property Φ has been violated

(condition $maxTrace$), or when there is no progress in the current iteration (which means deadlock).

The exploration process finishes when either the trace tree T_i is complete to depth $m+1$, expressed by $completelyExplored(T_i)$ or a maximum number of executions, $maxExecNr$ has been reached. The first case might not be feasible in practice.

4.4 Model Building

In this section, we will present the way a model is built once the exploration process stops. As earlier stated, our goal is to obtain a safe approximation of the real behaviour for all black box components in the system. Assuming a hypothetical precise model R_i for black box component C_i, a safe approximation of R_i is a model U_i that both underapproximates the controllable behaviour in R_i and overapproximates the uncontrollable behaviour.

Model U_i for black box component C_i will be built using the trace tree T_i associated to C_i. As T_i stores all positive and marks as such all negative sample traces tried during behaviour exploration, model U_i could, in a first instance, be built directly from tree T_i and minimized afterwards.

However, this would lead to losing a part of the information available in the tree structure T_i, such as the number of times a certain path t was observed at runtime, or whether after trace t event σ is unexplored, or found as not defining a valid transition. Therefore, trace tree T_i is first subject to a series of preprocessing that will be described later on, and only afterwards is model U_i built from T_i.

Directly building U_i from T_i would nevertheless lead to a very large, acyclic model U_i, which would remain acyclic even after it is minimized. Therefore, a very important precondition to building U_i is establishing which vertices in T_i actually correspond to the same state in the model.

4.4.1 Vertex Compatibility

Bounded Bisimulation

Consider two vertices v and v' in the trace tree T_i, so that $v, v' \in V^i \setminus \{\epsilon\}$. We say that v bisimulates v', and denote $v \sim v'$ if for each $\sigma \in \Sigma^i$ for which $\exists v'' \in V^i \setminus \{\epsilon\}. \xi^i(v, \sigma) = v''$ another vertex $v''' \in V^i \setminus \{\epsilon\}$ also exists so that $\xi^i(v', \sigma) = v'''$, and, symmetrically, for each $\sigma \in \Sigma^i$ for which $\exists v''' \in V^i \setminus \{\epsilon\}. \xi^i(v', \sigma) = v'''$, a vertex $v'' \in V^i \setminus \{\epsilon\}$ also exists so that $\xi^i(v, \sigma) = v''$, and the vertex v'' also bisimulates v'''. As the bisimulation relation is symmetrical, reflexive, and transitive, it is an equivalence relation.

Further on, we consider bisimulation on a depth of k. If vertices v and v' are bisimilar on a depth of 0, denoted $v \sim_0 v'$, they basically have outgoing transitions on the same set of events. If, also for all $\sigma \in \Sigma^i$ for which two vertices v'' and v''' exist so that $\xi^i(v, \sigma) = v''$ and $\xi^i(v', \sigma) = v'''$, the two vertices v'' and v''' are bisimilar on a depth of 0, $v'' \sim_0 v'''$, then we say that vertices v and v' are bisimilar on a depth of 1, which we denote $v \sim_1 v'$. If for all $\sigma \in \Sigma^i$ for which two vertices v'' and v''' exist so that $\xi^i(v, \sigma) = v''$ and $\xi^i(v', \sigma) = v'''$, vertices v'' and v''' are also bisimilar on a depth of 1, then v and v' are bisimilar on a depth of 2, etc. So, if all successors of vertex v are bisimilar on a depth of $k-1$ with the corresponding successors of vertex v', then v and v' are bisimilar on a depth of k, which we denote as $v \sim_k v'$.

We are interested on bisimulation on a depth of m, where m is the already introduced bound on an acyclic path. As both v and v' are vertices in a deterministic tree structure, thus both representing a root of a subtree, iff they are bisimilar on a depth of m, $v \sim_m v'$, then their underlying subtrees describe equivalent languages on traces of at most m events $\mathcal{L}(v) =_m \mathcal{L}(v')$. As any sequence of events of length greater than m contains at least one cycle, the two vertices v and v', bisimilar up to a depth of m, have the same language and thus correspond to the same state q in model U_i. This means that each pair of vertices v'', v''' that belong to the subtree of v and, respectively v' and have identical access traces from v, respectively v', also correspond to the same state q' in the model U_i, since U_i is deterministic.

Language Compatibility

For each vertex $v \in V^i \setminus \epsilon$, let us consider two languages of event traces starting from v $\mathcal{L}_{\surd}(v)$ and $\mathcal{L}_{\frown}(v)$, where $\mathcal{L}_{\surd}(v)$ is the language of observed traces $t \in \Sigma_*^i$ from v, while $\mathcal{L}_{\frown}(v)$ is the language of potential traces from v, thus including both observed and not yet tested traces $t' \in \Sigma_*^i$. Both $\mathcal{L}_{\surd}(v)$ and $\mathcal{L}_{\frown}(v)$ are regular languages and, furthermore, $\mathcal{L}_{\surd}(v) \subseteq \mathcal{L}_{\frown}(v)$.

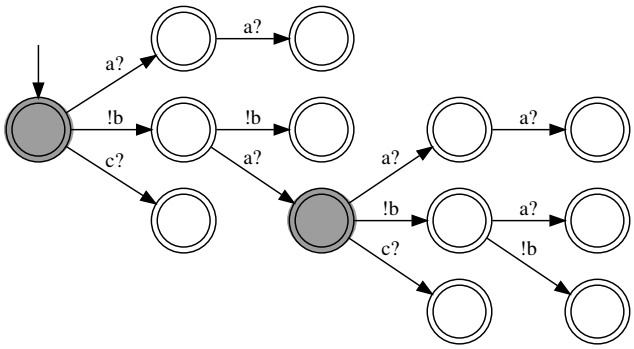
Vertices v and v' in the vertex set $V^i \setminus \{\epsilon\}$ are considered compatible, denoted as $v \simeq_\wr v'$, if the potential language of v' includes the observed language of v, and reciprocally $\mathcal{L}_{\surd}(v) \subseteq \mathcal{L}_{\frown}(v')$ and $\mathcal{L}_{\surd}(v') \subseteq \mathcal{L}_{\frown}(v)$. Thus, no trace observed from vertex v (i.e., following the access trace of v) is missing from the potential language of vertex v', and, reciprocally, no trace observed from v' was dismissed as a potential trace from v. This means the two vertices, v and v', could actually correspond to one and the same state q in the hypothetical precise model R_i of component C_i.

If this is true, then language $\mathcal{L}(q)$, the event trace language of state q in hypothetical model R_i, is included between languages $\mathcal{L}_{\surd}(v') \cup \mathcal{L}_{\surd}(v)$ and $\mathcal{L}_{\frown}(v) \cap \mathcal{L}_{\frown}(v')$. In this case, both the reunion of observed languages, and the intersection of potential languages converge to $\mathcal{L}(q)$ as more positive and negative samples, prefixed by the access trace of either v or v', are added to trace tree T_i.

The presented compatibility relation is commutative, so if $v \simeq_\wr v'$ is true, then $v' \simeq_\wr v$ is also true. As compatible vertices are assumed to correspond to the same state q in R_i, the relation of compatibility must be an equivalence relation, and thus, also transitive.

However, the compatibility relation, as introduced so far, is not transitive and cannot be used as an equivalence relation between the vertices in $V^i \setminus \{\epsilon\}$. Still, trace tree T_i can be adequately preprocessed so that relation $\simeq_\wr$ becomes transitive, and can be used as an equivalence relation over $V^i \setminus \{\epsilon\}$. After the preprocessing of T_i, the relation of compatibility between two vertices of T_i is reduced to a relation of bisimulation on a depth of m.

We present two variants of preprocessing the trace tree T_i: the one presented in section 4.4.2 is to be used when building an approximate model, while the one given in section 4.4.3, together with the consistency check at section 4.4.3, is used when we aim to build a complete model. The former transformation cuts $\mathcal{L}_{\frown}(v)$ down to a sublanguage of $\mathcal{L}_{\surd}(v)(\Sigma_!^i)*$ for all vertices $v \in V^i \setminus \{\epsilon\}$, while in



(a)

Figure 4.4: Two compatible vertices.

the latter case, the learning process continues and the traces in $\mathcal{L}_{\frown}(v) \backslash \mathcal{L}_{\surd}(v)$ are systematically tried at runtime, becoming either positive, or negative samples, until $\mathcal{L}_{\frown}(v) = \mathcal{L}_{\surd}(v)$.

After trace tree T_i is preprocessed, two vertices v, v' in the vertex set $V^i \setminus \{\epsilon\}$ are in a relation of language compatibility, $v \simeq_l v'$ if one of the following two conditions holds:

- vertices v and v' bisimulate each other up to depth m: $v \sim_m v'$
- if t is the access trace of v in tree T_i, so that $\xi^i_*(t) = v$, and t' the access trace of v', so $\xi^i_*(t) = v$, t and t' have a common suffix w, so that $t = uw$ and $t' = u'w$, where $u, u', w \in \Sigma^i_*$, and if prefixes u and u' are access traces for two vertices v'', respectively v''', then $v'' \sim_m v'''$ implies $v \simeq_l v'$.

In both cases, the compatibility relation between two vertices v and v' is reduced to a relation of bisimulation on a depth of m between v and v', $v \sim_m v'$, which is an equivalence relation. Thus, after the preprocessing is applied, in trace tree T_i, for vertices $v, v', v'' \in V^i \setminus \{\epsilon\}$, if $v \simeq_l v'$ and $v' \simeq_l v''$, then also $v \simeq_l v''$.

The set of vertices $V^i \setminus \{\epsilon\}$ of trace tree T_i can be partitioned, using a transitive vertex compatibility relation, into mutually exclusive subsets of compatible states. These compatibility partitions can be used later on to build the automaton U_i for black box component C_i.

Figure 4.4 presents an example of two compatible vertices in a trace tree, for which the parameter m is set to value 2. For simplicity, the empty vertices in the tree are not shown.

4.4.2 Building an Approximate Model

In order to build a safe approximate model U_i for component C_i using trace tree T_i, as stated earlier, we have to actually build, based only on the available information in T_i, an overapproximation of the real uncontrollable behaviour of C_i, and an underapproximation of its real controllable behaviour.

For this purpose, before partitioning the vertices in $V^i \setminus \{\epsilon\}$ in compatibility partitions and actually building U_i, we will have to preprocess some of the information trace tree T_i. This preprocessing will be described below.

For the preprocessing and model building phase, we introduce a special vertex ζ, the unknown-future vertex. If the empty vertex ϵ signified the observed fact that no transition on an event was possible, the unknown-future vertex constant ζ is always the target of one or more potential uncontrollable transitions, and signifies an unknown, possibly uncontrollable further development. No controllable events are further allowed from ζ, as the behaviour exploration process did not reach traces with this prefix. As ζ marks insufficient exploration, its presence, especially at small depths, makes part of component behaviour unusable in the composed system. Thus, using the vertex ζ is only a solution in the limit when there are too many traces to explore.

Pruning and Adding Behaviour

When the exploration process ends due to reaching the maximum number γ of allowed executions, thus leaving the black box component C_i incompletely explored, three final actions are applied on the associated trace tree T_i:

1. transitions on unexplored controllable events are removed from T_i – for every controllable event $\sigma \in \Sigma^i$, and for every vertex $v \in V^i \setminus \{\epsilon\}$, if $\xi^i(v, \sigma)$ is undefined, then $\xi^i(v, \sigma)$ becomes the empty vertex: $\forall v \in V^i \setminus \{\epsilon\}.\forall \sigma \in \Sigma^i_?.\ \nexists \xi^i(v, \sigma) \rightarrow \xi'^i(v, \sigma) = \epsilon$.

2. if a prefix in the trace tree was observed for a number of times that exceeds the fairness bound θ, all unobserved uncontrollable transitions are removed from its end vertex – for every uncontrollable event $\sigma \in \Sigma^i$, and for every vertex $v \in V^i \setminus \{\epsilon\}$, if $\xi^i(v, \sigma)$ is undefined, and the counter function $\nu(v) \geq \theta$, then $\xi^i(v, \sigma)$ becomes the empty vertex: $\forall v \in V^i \setminus \{\epsilon\}.\forall \sigma \in \Sigma^i_!.\ \nexists \xi^i(v, \sigma) \wedge \nu(v) \geq \theta \rightarrow \xi'^i(v, \sigma) = \epsilon$.

3. if a prefix in the trace tree was observed for less times than the established fairness bound, all unobserved uncontrollable transitions are added to its end vertex – for every uncontrollable event $\sigma \in \Sigma^i$, and for every vertex $v \in V^i \setminus \{\zeta\}$, if $\xi^i(v, \sigma)$ is undefined, and the counter function $\nu(v) < \theta$, then $\xi^i(v, \sigma)$ becomes the unknown-future vertex: $\forall v \in V^i \setminus \{\epsilon\}.\forall \sigma \in \Sigma^i_!.\ \nexists \xi^i(v, \sigma) \wedge \nu(v) < \theta \rightarrow \xi'^i(v, \sigma) = \zeta$.

Figure 4.5 presents the effects of pruning and adding behaviour to a trace tree, in a case when the parameter m is set to 2. The empty vertex is represented by the white rectangular-shaped nodes, while the unknown-future vertex is represented by the dark rectangular nodes. In figure 4.5(a) we can see the trace tree as it resulted from the exploration process, before any preprocessing being applied, then in figure 4.5(b), the unobserved controllable behaviour has been pruned out from the trace tree. Finally, in figure 4.5(c), we can see the trace tree after all infeasible uncontrollable behaviour has been pruned out, and all still feasible uncontrollable transitions have been added as transitions to the unknown-future vertex.

The reason for action 1 is that keeping infeasible controllable transitions in a model used to generate a controller for the system might lead to unsafe behaviour later on, in the composed system, since the adaptor/controller could try to enable an infeasible transition, fail, and thus block.

Action 2 is based on the fairness assumption made on the component, which we use to prune out infeasible uncontrollable events, i.e. the uncontrollable events that didn't occurred within the fairness bound. In model U_i, if an uncontrollable event σ hasn't occurred from state q, after the service has been in state q for more than θ times, with $visits(q) \geq \theta$, then we conclude that σ never actually occurs from q, and the transition on σ from q can be removed.

Action 3 treats the case when a state q in the model U_i has been visited for less than θ times. Since the fairness bound hasn't been reached, from state q all uncontrollable transitions are still possible, and none of them can be cut out. Furthermore, we cannot predict the further development of event traces after one of these unobserved controllable events would occur. Therefore, under the goal of overapproximating uncontrollable behaviour in the model, the only assumption we can make is the most pessimistic one, that any of these hypothetical transitions would lead the execution into a state from which all uncontrollable events can occur, all triggering self-cycle transitions back to the same state. Therefore, all unobserved uncontrollable transitions are added to

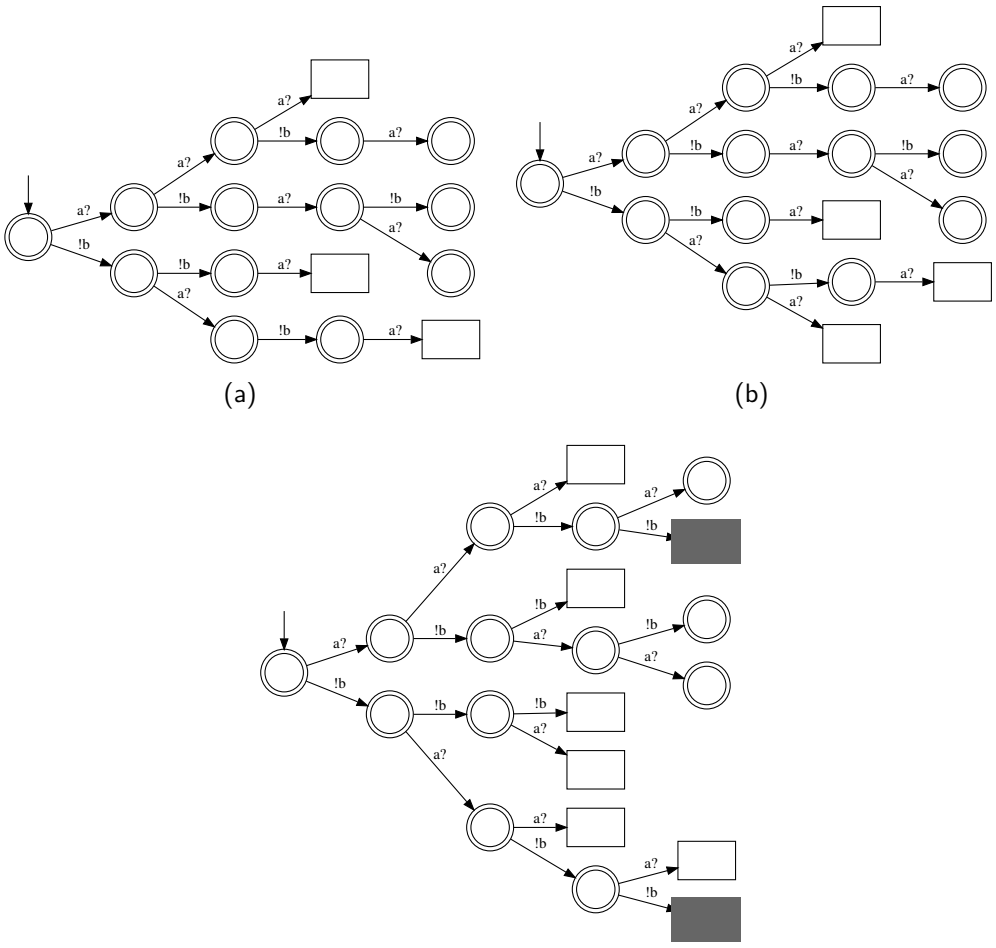


Figure 4.5: Pruning and adding behaviour to the trace tree: 4.5(a) – the trace tree before preprocessing, 4.5(b) – trace tree after action 1, 4.5(c) – trace tree after actions 2 and 3.

the model, from the incompletely explored state to a special state with self-loops on all uncontrollable events and no controllable outgoing transitions.

Thus, each learned model is characterized by an overapproximation of the real uncontrollable behaviour and an underapproximation of the controllable one, which allows for a reliable adaptor synthesis – if an an adaptor can be obtained starting from these models, it will also work on the real system.

Choosing the Fairness Bound

A problem that would arise in this context would be how to properly choose the fairness bound θ when learning a black box component C_i. A well chosen fairness bound should be permissive enough to overapproximate the real fairness bound of the system, and thus let all uncontrollable events feasible from a state q occur at some point from q, without leading to a premature cut out of some unobserved uncontrollable events. Also, such a fairness bound should also be tight enough to actually be useful in pruning out infeasible uncontrollable events, since a fairness bound θ that is too loose would only lead to a final model that is actually impossible to control.

To properly identify such a bound θ for black box C_i, we have turned to a classic probability theory problem: the coupon collector's problem [66]. Basically, the coupon collector's problems resides in the following. Given N different coupons, equally likely, which are being collected with replacement, and a collector willing to obtain them all, how many times X does the collector have to draw a coupon, with replacement, before having drawn each coupon at least once?

As shown in the literature, the estimated number of trials noted as $E[X]$, is asymptotically converging, when $n \to \infty$, to

$$E[X] = n\ln n + \gamma' n + \mathcal{O}(1)$$

where $\gamma' \approx 0.5772156649$ is the Euler – Mascheroni constant.

For simplicity, this can be further expressed as:

$$E[X] = n\ln n + \mathcal{O}(n)$$

Let us assume the number of uncontrollable events in the event set Σ^i corresponding to black box component C_i is $|\Sigma^i_!| = n^i_!$. We take into account the

fact that from a same state q we can have both controllable and uncontrollable events, and thus, we assume that only in half of the times state q is reached an uncontrollable event occurs. Thus, using the result above, we set the fairness bound θ_i for component C_i to:

$$\theta_i = 2n^i_! \mathtt{ln} n^i_!$$

If, for convenience, we want to use only one fairness bound θ for the whole system, instead of a θ_i for each component, then we will choose the maximum θ_i as the global fairness bound θ. Thus, $\theta = max(\theta_0, \theta_1, .., \theta_{n-1})$, which means $\theta = 2n_! \mathtt{ln} n_!$, where $n_! = max(n^0_!, n^1_!, .., n^{n-1}_!)$ is the maximum number of uncontrollable events. Therefore, the chosen global fairness bound θ is:

$$\theta = 2n_! \mathtt{ln} n_!$$

Building the Model

The model U_i is built from the trace tree T_i in the following way. First, after the behaviour exploration process stops, the unobserved controllable and infeasible uncontrollable traces are pruned out from the trace tree, as showed above. Then, starting with the root vertex v^i_0 of trace tree T_i, and continuing with vertices at depth 1, then at depth 2, etc. and finishing with vertices at depth m, each vertex is checked for compatibility with the rest of the vertices in the model. During this phase, compatibility-based partitioning takes place, and thus, based on the relation of compatibility between vertices, the vertex set $V_i \setminus \{\epsilon, \zeta\}$ is partitioned into a set of compatibility partitions $P_{\simeq}(V_i \setminus \{\epsilon, \zeta\}) = \{P_0, P_1, ..., P_{np-1}\}$, which we also denote by $P^i_{\simeq}$.

From the number np of partitions in $P^i_{\simeq}$ we then obtain the total number of states of model U_i, which is $np + 1$, because of adding q_ζ, the state that corresponds to unknown-future vertex ζ.

Then, we start building the approximate model U_i for black box component C_i. The set of states Q^i is $Q^i = \{q_0, q_1, ..., q_{np-1}\} \cup \{q_\zeta\}$, where each state q_j corresponds to a compatibility partition P_j in partition set $P^i_{\simeq}$. We denote the correspondence between a state q_j in U_i, and its associated partition P_j as $q_j \asymp P_j$. The initial state q_0 of model U_i is implicitly associated to partition P_0, by which we specifically denote the partition that includes the root vertex v_0 of trace tree T_i, thus $q_0 \asymp P_0$ and $v_0 \in P_0$.

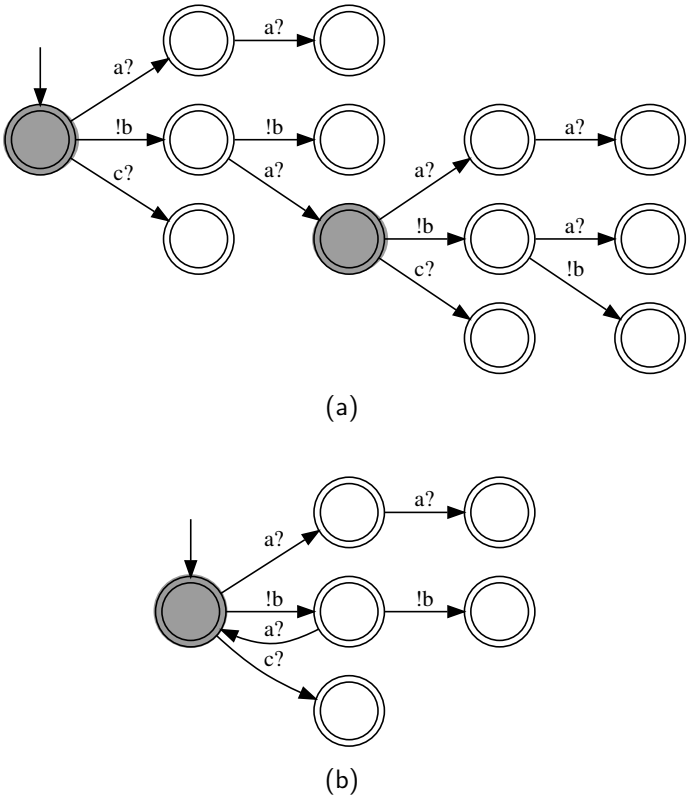
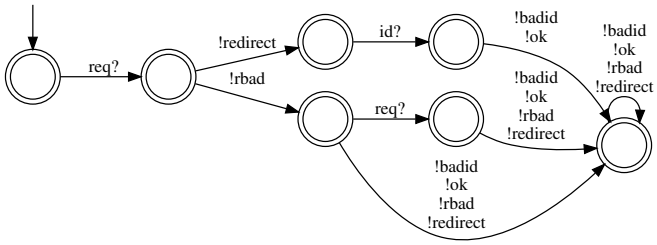


Figure 4.6: Two compatible states in the trace tree – figure 4.6(a), lead to one state in the built model – figure 4.6(b).



(a)

Figure 4.7: Model with transitions towards an unknown-future state.

By a breadth-first traversal of trace tree T_i, the transition function δ^i of model U_i, defined as $\delta^i : Q^i \times \Sigma^i \to Q^i$ is built. This happens as follows. Let v be the currently analyzed vertex in T_i, where v is in partition P, and q is the state in Q^i associated to P, $q \asymp P$. For each event $\sigma \in \Sigma^i$, such as $\xi^i(v, \sigma) \in V^i \backslash \{\epsilon\} \cup \{\zeta\}$, let vertex $v' = \xi^i(v, \sigma)$. Then, we have two possible cases:

- $v' \neq \zeta$: if v' is a regular vertex, then let q' be its corresponding state in Q^i so that $q' \asymp P'$ and $v' \in P'$; the transition function δ^i of model U_i then becomes $\delta'^i(q, \sigma) = q'$.
- $v' = \zeta$: if v' is the unknown-future vertex, then the transition function δ^i of model U_i will be enriched with a transition on σ to unknown-future state q_ζ, thus becoming $\delta'^i(q, \sigma) = q_\zeta$.

In figure 4.6 we can see how two compatible traces in a trace tree (figure 4.6(a)), end up as the same state in the built model, and how this actually creates a cycle in the built model(figure 4.6(b)).

The unknown-future state q_ζ is defined as having self-loops on all uncontrollable events $\sigma \in \Sigma^i_!$, and no outgoing controllable transitions, thus $\forall \sigma \in \Sigma^i_!.\ \delta^i(q_\zeta, \sigma) = q_\zeta$ and $\forall \sigma \in \Sigma^i_?.\ \nexists \delta^i(q_\zeta, \sigma)$.

An example of a built model that contains an unknown future state cand be seen in figure 4.7.

4.4.3 Building a Precise Model

In this section, we will present the algorithm for building a precise model for a black box component C_i. We assume that the behaviour of component C_i has

been completely explored up to depth $m+1$, and, thus, in the trace tree T_i, all vertices $v \in V_i \setminus \{\epsilon\}$ are completely explored, $complete(v)$, iff their access trace $t \in \Sigma^i_*$ s.t. $\xi^i_*(t) = v$ has $m+1$ or less events: $|t| \leq m+1$.

Also, we assume that the maximum number of executions γ hasn't been reached, and that, if necessary, the behaviour of the component C_i can be further explored by resuming the exploration with new traces.

Pruning Infeasible Behaviour

Since trace tree T_i is complete up to the depth of $m+1$, $complete(T_i, p)$, this means all traces $t \in \Sigma^i_*$ of length $|t| \leq m+1$, for which the string transition function returns a valid vertex $\xi^i_*(t) \in V^i \setminus \{\epsilon\}$, have been observed for at least θ times. Thus, for all completely explored vertices v, since $\nu(v) \geq \theta$, the unobserved uncontrollable events are considered as infeasible and cut out.

Similarly to action 2 described in subsection 4.4.2, for every uncontrollable event $\sigma \in \Sigma^i$, and for every vertex $v \in V^i \setminus \{\epsilon\}$, if $\xi^i(v, \sigma)$ is undefined, and the counter function $\nu(v) \geq \theta$, then $\xi^i(v, \sigma)$ becomes the empty vertex: $\forall v \in V^i \setminus \{\epsilon\}. \forall \sigma \in \Sigma^i_!. \nexists \xi^i(v, \sigma) \wedge \nu(v) \geq \theta \rightarrow \xi'^i(v, \sigma) = \epsilon$.

However, it is only the infeasible uncontrollable behaviour that is pruned out. Unobserved and untried controllable behaviour is not removed from the trace tree in this case, but it will be further explored.

Control Consistency Check

After pruning out the infeasible uncontrollable behaviour from trace tree T_i, the next phase requires that each vertex is checked for compatibility with the rest of the vertices in the model, and then compatibility-based partitioning of the vertex set V^i takes place.

Then, the set of compatibility partitions $P_{\simeq}(V^i \setminus \{\epsilon\}) = \{P_0, P_1, ..., P_{np-1}\}$ results, also denoted as $P^i_{\simeq}$. Consider a partition $P_j \in P^i_{\simeq}$, containing a finite number of vertices nv, thus $P_j = \{v_{j_0}, v_{j_1}, ..., v_{j_{nv}}\}$. Although the vertices v_{j_k} have no unexplored transitions, they might not be completely explored up to the depth of m.

If state q_j will correspond to partition P_j in model U^i, $q_j \asymp P_j$, its language $\mathcal{L}(q_j)$ will include all the languages of all vertices v_{j_k}, $\bigcup \mathcal{L}(v_{j_k})$ in the partition. Thus,

we need to make sure that the vertices can be indeed treated as equivalent.

For simplicity, let us assume we have two vertices, v' and v'' in the same compatibility partition P_j. We also assume that a certain trace t belongs to language $\mathcal{L}(v')$, but not to language $\mathcal{L}(v'')$, $t \in \mathcal{L}(v') \wedge t \notin \mathcal{L}(v'')$. Let us also assume $t = u\sigma w$, where $u, w \in \Sigma_*^i$ and $\sigma \in \Sigma^i$, so that $u \in \mathcal{L}(v'')$, but $u\sigma \notin \mathcal{L}(v'')$. We call such a trace $u\sigma$ a potentially separating sequence.

Then, if trace t'' is the access trace for vertex v'', $\delta_*^i(t'') = v''$, then the black box C_i has to be tested for trace $t''u\sigma$, the separating test sequence, which is formed by concatenating the potentially separating sequence to the access trace of v'', in order to decide whether vertices v' and v'' can be considered equivalent. If event σ cannot be enabled after trace $t''u$, then it means that vertices v' and v'' are no longer compatible, and therefore they don't belong in the same compatibility partition P_j. This requires a splitting of partition P_j into two partitions P_j' and P_j'', by determining for each vertex v in P_j whether it is compatible with vertex v' or with v''.

However, since such situations might appear for several vertices in $V^i \backslash \{\epsilon\}$, which would further complicate a partition splitting solution, the chosen solution is different. Once all potentially separating sequences have been tried at runtime, we simply redo the equivalence partitioning for all vertices in trace tree T_i.

Thus, in any compatibility partition P_j, for all pairs of vertices v', v'', for which a potentially separating sequence exists for one vertex, a separating test sequence can be easily derived for the other and tried at runtime. When all potentially separating sequences have been tried, the equivalence partitioning is redone. If new potentially separating sequences are then found for the new partitions, they are again tested, etc. This consistency enforcing phase continues until no potentially separating sequence is found for any pair of vertices in any partition P_j of the vertex set V^i.

If the maximum number of executions is reached before the consistency testing ends, then the trace tree is preprocessed using the metod in section 4.4.2, the compatibility partitions are recomputed and the model is built using the algorithm for building an approximate model, as shown in subsection 4.4.2, .

Building the Model

The precise model U_i is built from the complete trace tree T_i as follows. After the behaviour exploration is complete to depth $m + 1$, infeasible controllable

traces are pruned out from the trace tree, as showed above. Then, starting with the root vertex v_0^i of trace tree T_i, and continuing with vertices at depths from 1 to m, each vertex is checked for language-based compatibility with the rest of the vertices in the model.

Thus, language-based compatibility partitioning takes place over the vertex set $V_i \setminus \{\epsilon\}$, obtaining the set of compatibility partitions $P_{\simeq}(V_i \setminus \{\epsilon\})$, or $P_{\simeq}^i$.

The precise model U_i for black box component C_i is built as following. The set of states Q^i is $Q^i = \{q_0, q_1, ..., q_{np-1}\}$, where each state q_j corresponds to a compatibility partition P_j in partition set $P_{\simeq}^i$, $q_j \asymp P_j$. This means that model U_i will have np states. The initial state q_0 of model U_i is implicitly associated to partition P_0, the partition including the root vertex v_0 of trace tree T_i, thus $q_0 \asymp P_0$ and $v_0 \in P_0$.

By a breadth-first traversal of trace tree T_i, the transition function δ^i of model U_i, defined as $\delta^i : Q^i \times \Sigma^i \to Q^i$ is built. This happens as follows. Let v be the currently analyzed vertex in T_i, where v is in partition P, and q is the state in Q^i associated to P, $q \asymp P$. For each event $\sigma \in \Sigma^i$, such as $\xi^i(v, \sigma) \in V^i \setminus \{\epsilon\}$, let vertex $v' = \xi^i(v, \sigma)$. Then, if v' is a regular vertex, then let q' be its corresponding state in Q^i so that $q' \asymp P'$ and $v' \in P'$; the transition function δ^i of model U_i then becomes $\delta'^i(q, \sigma) = q'$.

4.4.4 Model building algorithms

In the following, we present the algorithms for both approximate and precise model building, in a concise pseudocode form.

Approximate model building algorithm

The algorithm for building an approximate model is presented below. As previously stated, in the **approxBuild** procedure, the trace tree T_i is first preprocessed by pruning out all behaviour found as infeasible (procedure **pruneInfeasible**), and by adding to it all uncontrollable behaviour that is still feasible (procedure **addFeasible**). Then, based on the compatibility relations defined at section 4.4.1, the vertices in the vertex set V^i, associated to the trace tree T_i, are partitioned into compatibility sets. Using the partition set P_i as a reference for the state set Q^i, procedure $build$ then builds approximate model U_i.

Algorithm 3 : Approx. model building

1: **procedure** approxBuild(U_i, T_i)
2: pruneInfeasible(T_i)
3: addFeasible(T_i)
4: P_i =partition(T_i)
5: build(U_i, T_i, P_i)

The mentioned procedures are then described more in detail. Predicate **infeasible** defines when an unobserved event σ is considered infeasible from a vertex v in trace tree T_i: if either it is a controllable event, or an uncontrollable event and the trace access of vertex v has been observed for more than θ times, where θ is the fairness bound.

Predicate **compatible** describes the condition for two vertices v and v' to be compatible in trace tree T_i.

Procedure **pruneInfeasible**, based on the evaluation of the infeasibility condition, cuts out an unobserved event σ from a certain vertex v by adding a link from v, on σ, towards the empty vertex ϵ. It does so for all pairs (v, σ) of events in the event set Σ^i and vertices $v \in V^i$ for which predicate **infeasible**(v, σ) evaluates to true.

Somehow similar, procedure **addFeasible**, adds to trace tree T_i links towards the unknown-future vertex ζ for all unobserved uncontrollable events that are still considered feasible from a vertex v, where v has been reached, at runtime, for less than θ times.

The vertex partitioning function, **partition**, works as follows. Initially, each partition in the partition set contains only one vertex, so we have as many partitions as vertices are in V^i at a depth of at most m. Then, all pairs of vertices v and v' in V^i, at a depth less than, or equal to m, are checked for compatibility as according to the described predicate **compatible**(v, v'). If two different vertices are found compatible, their corresponding compatibility partitions are merged. In the end, the final partition set is returned.

Algorithm 4 : Auxiliary procedures

def infeasible$(v, \sigma) = \nexists \xi^i(v, \sigma) \wedge (\sigma \in \Sigma^i_? \vee (\sigma \in \Sigma^i_! \wedge \nu(v) \geq \theta))$
def compatible$(v, v') = (\mathcal{L}_\surd(v) \subseteq \mathcal{L}_\sim(v')) \wedge (\mathcal{L}_\surd(v') \subseteq \mathcal{L}_\sim(v))$

procedure pruneInfeasible(T_i)
for all $v \in V^i$ **do**
  for all $\sigma \in \Sigma^i$ **do**
    if infeasible(v, σ) **then**
      $\xi^i(v, \sigma) = \epsilon$

procedure addFeasible(T_i)
for all $v \in V^i$ **do**
  for all $\sigma \in \Sigma^i_!$ **do**
    if $\nexists \xi^i(v, \sigma) \wedge \nu(v) < \theta$ **then**
      $\xi^i(v, \sigma) = \zeta$

function partition(T_i)
$P_i = \emptyset$
for all $v \in V^i \wedge depth(v) \leq m$ **do**
  $P = \{v\}$
  $P_i = P_i \cup \{P\}$

for all $v \in V^i$ **do**
  for all $v' \in V^i . v' \neq v$ **do**
    if compatible$(v, v') \wedge (depth(v) \leq m) \wedge (depth(v') \leq m)$ **then**
      $P = matchVertexToPart(v, P_i)$
      $P' = matchVertexToPart(v', P_i)$
      $mergePartitions(P, P')$
return P_i

Algorithm 5 : Build model from trace tree

```
1: procedure build (U_i, T_i, P_i)
2: initialize(U_i)
3: Q^i = partToStates(P_i)
4: recursiveBuild(v_0, 0, i)
5:
6: procedure recursiveBuild(v, depth, i)
7: P = matchVertexToPart(v, P_i)
8: q = matchStateToPartition(P)
9: if depth ≤ m then
10:   V = v.childNodes
11:   for all v' ∈ V do
12:     σ = v.eventTo(v')
13:     P' = matchVertexToPart(v', P_i)
14:     q' = matchStateToPartition(P)
15:     trans = makeTransition(q, σ, q')
16:     U_i.addTransition(trans)
17:     recursiveBuild(v', depth + 1, i)
```

Finally, by means of procedure **build**, the model U_i is being constructed for component C_i using the trace tree T_i and the computed partition set P_i. Each vertex partition in the partition set P_i corresponds to a state in the state set Q^i of model U_i.

The trace tree T_i is traversed recursively by procedure **recursiveBuild**, and from each vertex $v \in V^i$, if a valid link on an event σ exists, leading to a non-empty vertex $v' \in V^i \cup \{\zeta\}$, a transition in U_i is created as follows. Partitions P and P' in partition set P_i, for which $v \in P$ and $v' \in P'$, are identified and so are their corresponding states $q, q' \in Q^i$. If v' is vertex ζ, then q' is the unknown-future state q_ζ. Then, a transition on event σ from q to q' is created in U_i.

Further on, the building procedure continues recursively if $v' \in V^i$, using v' as the new start vertex in the recursive call. The maximum depth up to which a vertex v is recursively explored by the **recursiveBuild** procedure is m.

Precise model building algorithm

The algorithm for building a precise model for black box component C_i, using trace tree T_i, is described below. When a precise model is being built, we assume that the maximum number of executions hasn't yet been reached, and so the black box C_i can still be explored, if necessary.

Algorithm 6 : Auxiliary procedures

procedure pruneInfeasUnctrl(T_i)
for all $v \in V^i$ **do**
  for all $\sigma \in \Sigma^i_!$ **do**
    if $\nexists \xi^i(v,\sigma) \wedge \nu(v) \geq \theta$ **then**
      $\xi^i(v,\sigma) = \epsilon$

procedure consistCheck(T_i, P_i)
for all $P \in P_i$ **do**
  for all $v \in P$ **do**
    for all $v' \in P$ s.t. $v \neq v'$ **do**
      $\mathcal{L} = \mathcal{L}_\surd(v) \setminus \mathcal{L}_\surd(v')$
      $\mathcal{L}' = \mathcal{L}_\surd(v') \setminus \mathcal{L}_\surd(v)$
      if $\exists t \in \mathcal{L}.t = u\sigma w \wedge u \in \mathcal{L}_\surd(v') \wedge \sigma \in \Sigma^i_?$ **then**
        $s = accessTrace(v')$
        $mismatch = mismatch \vee \neg test(st, C_i)$
      if $\exists t \in \mathcal{L}'.t = u\sigma w \wedge u \in \mathcal{L}_\surd(v) \wedge \sigma \in \Sigma^i_?$ **then**
        $s = accessTrace(v)$
        $mismatch = mismatch \vee \neg test(st, C_i)$

Algorithm 7 : Precise model building

procedure preciseBuild(U_i, T_i)
repeat
  $mismatch = false$
  pruneInfeasUnctrl(T_i)
  P_i =partition(T_i)
  consistCheck(T_i, P_i)
until $\neg mismatch$
build(U_i, T_i, P_i)

First, the trace tree T_i is preprocessed in a slightly different way that when building approximate models. Thus, only infeasible uncontrollable events are pruned out, by procedure **pruneInfeasUnctrl**, since it is assumed that the controllable events have been exhaustively explored up to depth $m + 1$.

Then, the vertex set is partitioned by the **partition** function, and the resulting partition set is then checked for consistency by procedure **consistCheck**, which may lead to some further testing on black box component C_i if any differentiating sequences are found between any two vertices of the same partition (marked by a value of $true$ for variable $mismatch$). The procedures **pruneInfeasUnctrl**, **partition** and **consistCheck** are repeatedly invoked in this order, until no new differentiating sequence is found. Only after this phase is completed, the behavioural model U_i is built for black box C_i by means of the **build** procedure.

The **partition** procedure and the **build** procedure are similar to the ones used by the approximate model building algorithm, previously presented, so we don't provide further details here with respect to them.

An important role is played by the consistency check procedure **consistCheck**. Its role is to verify that all compatible vertices are actually equivalent – since we aim to obtain a precis model, and so, if a controllable trace can be produced from a vertex v, this trace can also be obtained from all vertices in the same compatibility partition.

Therefore, each time that for a pair of vertices v, v' from the same compatibility partition P, an observed trace t is found that only belongs to the observed language $\mathcal{L}$ of one of the two vertices, say v, then t has to be also tried from v'. To do this, the access trace s of v' is obtained with operation $accessTrace$, and the differentiating trace t is concatenated to it. The resulting sequence is used to test black box C_i, as $test(st, C_i)$. If the test is successful, then no action is taken. Otherwise, the two vertices v and v' have been proven as non-equivalent, and repartitioning of vertex set V^i is needed.

To save effort, this repartitioning is only performed after all vertices in partition P have be analyzed and had their possible incompatibilities tested, and it is done separately, in the main loop of algorithm 7, after a new pruning of infeasible uncontrollable transitions.

4.5 Distributed Behaviour Exploration

4.5.1 Motivation

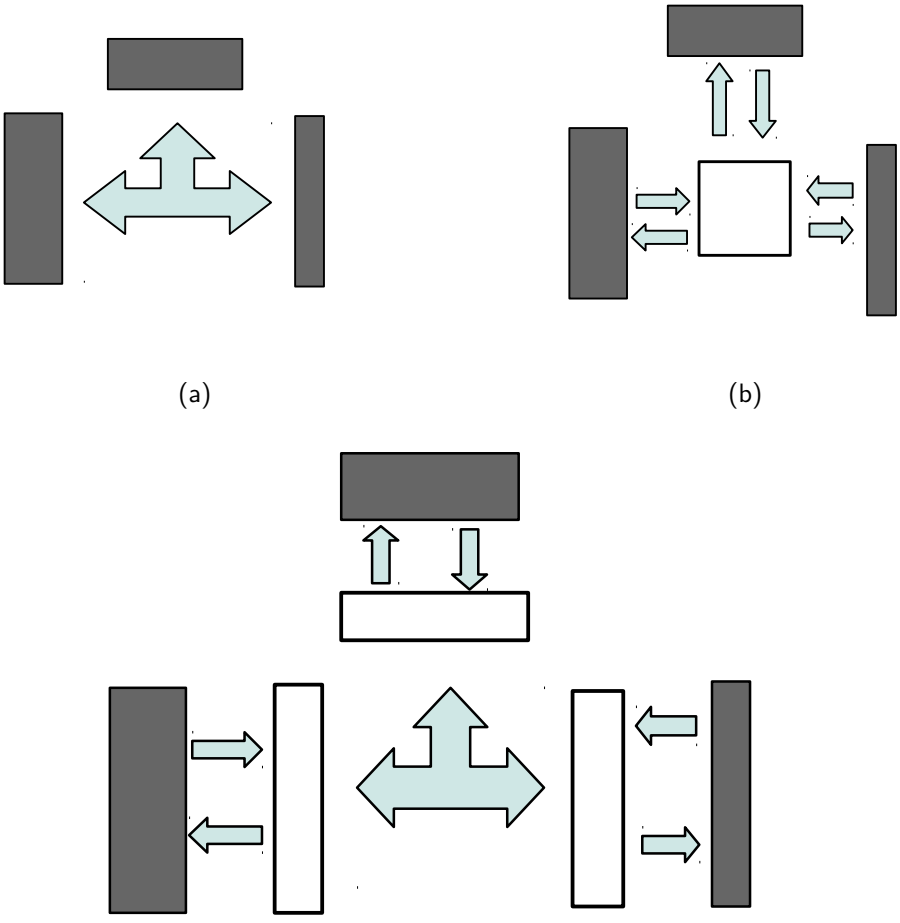
Exploring the component behaviour in a centralized manner, with all components being executed together, is expensive. For a system with n black box components that is explored at execution in a centralized way, the total length of a maximum execution trace often exceeds $2mn$. A longer execution trace involves more input events, i.e. forwarding more messages to the component, and each such input event can have a non-negligible cost. However, if the behaviour exploration is localized for each black box component in the system, the execution traces used for local learning contain at most $2m$ events, and, moreover, the n instances of the model inference process work in parallel, thus resulting in a more efficient learning.

When remote components are considered, inferring models locally corresponds best to the natural structure of the distributed system. Further on, there are system specifications which can be met in a decentralized way, by building only local, instead of centralized controllers. For such systems, this approach is best suitable, since local model learning leads straightforwardly to adaptor generation, without the need of knowing behavioural models for remote components. However, in our approach, only the model learning phase is local so far, and the controller obtained for the system is still centralized.

4.5.2 Method overview

Let us assume that for each remote component we have a local proactive adaptor, that monitors and controls only the behaviour of its corresponding component. This local proactive adaptor intercepts the messages sent by the component and it broadcasts them over the network. The messages are broadcast in a non-redundant way, i.e., the local adaptor won't broadcast the same message twice, unless it's necessary. The sent messages are received by other local adaptors, which can store them and forward them locally, whenever needed to explore the behaviour of their corresponding local component.

Assume that all n components in the system S are remote components, and each has its own, local proactive adaptor that observes and controls the component locally. This results in n local adaptors LM_i, $i = \overline{0, n-1}$. Each local adaptor



(a) (b)

(c)

Figure 4.8: Learning Setup: 4.8(a) before: components interact unrestricted, 4.8(b) centralized: all interactions take place through the proactive adaptor in the middle. 4.8(c) localized: each component has its own proactive adaptor, and interacts with it locally.

LM_i monitors only the local component C_i, and communicates with the other adaptors by resending the messages sent by C_i and receiving messages sent by the other components C_j, $j \neq i$. Also, as soon as component C_i sends a message msg, adaptor LM_i intercepts it and will broadcast it to all adaptors $LM_j, j \neq i$, whose associated components C_i have the corresponding controllable event $\sigma = msg?$ in their alphabet $\sigma \in \Sigma^i$.

The behaviour exploration is performed by the local adaptor on the monitored and controlled component similarly to the centralized case when all other components would be known. This means that the local intelligent adaptor knows the desired property for the system, and a precise or most general model for each of the other components, that are remote from its point of view. We define a most general model for a component C_i as a model that has only one state, and has self-loop transitions from that state on all events $\sigma \in \Sigma^i$. Thus, the language described by a most general model is Σ_*^i.

The local black box component C_i is associated, as described in section 4.3, to a trace tree T_i, that contains all runtime observations regarding the monitored and controlled behaviour of C_i.

The current state of all remote component models and of the system specification automaton is continuously updated during runtime, in order to match the current execution. The trace tree associated with the local black box component is augmented with new information each time new runtime behaviour is observed, i.e. new execution traces are tried, becoming positive or negative samples, the number of times an access trace is observed reaches the fairness threshold θ, thus making unobserved uncontrollable events infeasible from its associated vertex, etc.

Only behaviour relevant to the desired property is explored, and this happens by means of verification-driven execution for the local black box component, and by means of a depth-first search in the models of the remote components. In order to enable controllable, i.e. receive transitions in the local component the proactive adaptor forwards one or another of the messages received from remote adaptors. These are actual messages sent at some point during their separate, individual exploration process by the remote components. Based on how the real local component reacts to the forwarded messages, the sequence formed as the current execution trace suffixed by the enabled receive event is classified as a positive or negative sample, and marked as such in the local trace tree.

Probably the most important difference between the centralized and the dis-

tributed approach resides in the way components interact during the exploration process: if in the centralized approach the real components are explored together, in the distributed case each component is explored individually, together with only the models corresponding to the other components. These models can be precise, if the remote component is well-specified, or are assumed as most general, when the remote component is also a black box. Thus, even if the remote black box component has its model inferred in the meanwhile, the local exploration process has no access to the remotely acquired information, using a most general model instead.

Therefore, while allowing for parallelism in the exploration process, the distributed method has the disadvantage that the local component might be explored against execution scenarios that are actually infeasible for the real system, and thus useless for the composition goal. This can happen because most permissive, general models are used for remote black boxes, and exploring these models by a bounded depth-first search can result in simulated system interactions that would never occur between the real components.

4.5.3 Local Exploration Strategy

The behaviour of each component C_i is locally controlled by a proactive adaptor LM_i. Each such adaptor intercepts the messages sent by the component C_i, and forwards it messages that were previously emitted by the other components in the system plant. The adaptors communicate among themselves in the sense that messages sent by C_i and locally intercepted by its adaptor are forwarded to the rest of the adaptors LM_j, $j \neq i$.

Let us first assume that we are not dealing with black boxes, that all component models are known, and denoted as M_i. We also denote by $M_\times$ the asynchronous product of all component models $M_\times = M_0 \times M_1 \times ... \times M_{n-1}$. If component C_i is the local component, then its context E_i is represented by the asynchronous product $E_i = M_0 \times ... \times M_{i-1} \times M_{i+1} \times ... \times M_{n-1}$, together with the safety property Φ. Then, we can decompose the product between the plant and the safety property as the asynchronous product between the safe context behaviour and the safe component behaviour: $M_\times \| \Phi = (E_i \| \Phi) \times (M_i \| \Phi)$.

In our learning setting, however, we have at least one black box component, whose model is inferred. Still, the decomposition presented above holds and will be used in the local model learning solution we present in this section.

Controlling a Current Execution

In order to describe the process by which a current execution is controlled, we first assume the case of a component whose model has been provided. However, in the actual learning setting, this component is a black box and the execution is controlled with the purpose of exploring its behaviour and learn a model of this behaviour.

So, let us assume M_i is a known model of a component C_i, which is locally monitored and controlled by LM_i. During an execution the exploration of model M_i is always matched with the actual execution of locally monitored component C_i. Also, each model M_j, where $j \neq i$, describes the behaviour of a remote component C_j, which is beyond the control of the local proactive adaptor. Thus, for every remote component C_j only its model M_j can be explored, and this happens by means of a bounded depth-first search.

Suppose that, while monitoring component C_i, from a state q in M_i an uncontrollable event σ occurs at runtime, i.e. a message sent by the component C_i is intercepted by the local adaptor LM_i. Then, if a transition triggered by σ exists in the property automaton Φ, from its current state q^{Φ}, automaton Φ is advanced by that transition, and its new current state becomes $\delta(q^{\Phi}, \sigma)$. Otherwise, if no such transition exists, the property Φ is violated at runtime and the current execution stops.

As defined in section 4.3, a control point in a model M_i is a state q_{cp} for which the controllable event set $\Sigma^i_?(q_{cp})$ contains 2 or more receive events: $|\Sigma^i_?(q_{cp})| \geq 2$. This means that in such a state q_{cp} we can choose from at least two possible execution paths for component C_i.

When the execution reaches a control point q_{cp}, a receive event $\sigma \in \Sigma^i_?(q_{cp})$ is enabled for execution if from the current state of Φ a transition on σ exists. The receive event σ is enabled by forwarding its associated expected message to component C_i. If no such message is available, then model M_i and specification automaton Φ cannot be advanced, as the current run does not progress.

In this case, the modelled safe environment $E_i \| \Phi$ is explored by a bounded depth-first search phase, that tries to advance property automaton Φ to a state from which a different set of controllable transitions are allowed in M_i. Then, we can forward to C_i other messages than we were previously allowed to. If still no progress is made, then we have reached a deadlock and the current execution must be reset.

The execution is forced to end by resetting C_i when either the property Φ is violated by the local component, or deadlock has been detected or when the execution trace has reached its maximum allowed length.

Local Black Box Exploration

However, no model M_i is available for the local black box component. Thus, similarly to section 4.3 we use the property automaton Φ to generate test sequences, and we use a trace tree structure T_i, defined at section 4.2.2 to establish priorities between different sequences, to store positive and negative samples and to define a termination condition for the learning process.

As in the centralized case, the black box component C_i is associated to a trace tree T_i. Let us assume the current execution trace is t, and that it has led the property automaton Φ to a state q. Let $t_{|i}$ be the projection of trace t on the event set Σ^i, corresponding to component C_i.

Assume an event $\sigma \in \Sigma^i$ occurs and that in property automaton Φ, the transition on σ from the current state exists and is taken. Then, the proactive adaptor LM_i intercepts the sent message and broadcasts it to the other proactive adaptors in the learning setting LM_j, $j \neq i$.

After this, the current execution trace becomes $t\sigma$. If the projection trace $t_{|i}\sigma$ has never been tried at runtime before, it has to be marked as observed in the tree T_i. Let $v = \xi^i_*(t_{|i})$ be the vertex returned by the string transition function of T_i. If $t_{|i}\sigma$ hasn't been observed before, $\xi^i(v, \sigma)$ is still undefined. Therefore, a new vertex v' has to be added to V^i, so that the transition function ξ^i changes to $\xi^i(v, \sigma) = v'$, while $\xi^i(v', \sigma')$ will be undefined for the moment, for all $\sigma' \in \Sigma^i$. In rest, the transition function ξ^i stays the same. Also, the counter function for v' becomes $\nu(v') = 1$, while remaining unchanged in rest.

If, however, $t_{|i}\sigma$ is a trace already in T_i, and vertex v' so that $v' = \xi^i_*(t_{|i}\sigma)$ already exists, then the only thing that changes is the counter function, since trace $t_{|i}\sigma$ has been observed once more. Therefore, the result returned by the counter function for vertex v' increases with 1: $\nu_{new}(v') = \nu_{old}(v') + 1$.

Let us now assume the current execution trace is t, and that property automaton Φ is in a current state q, which is also a control point. Also assume that from q, in Φ, we can choose between controllable events σ and σ', which are both in the event set Σ^i, of component C_i. We now have to decide which of the two

events to enable in C_i, and in doing so we have three possible cases, as earlier mentioned in section 4.3:

- none of the traces $t_{|i}\sigma$ and $t_{|i}\sigma'$ has been previously tried: $untried(t_{|i}\sigma) \wedge untried(t_{|i}\sigma')$, so we randomly choose which event to enable
- only the local projection trace $t_{|i}\sigma$ has been previously tried: $(positive(t_{|i}\sigma) \vee negative(t_{|i}\sigma)) \wedge untried(t_{|i}\sigma')$, so it is event σ' which is enabled, as we want to cover as much unexplored behaviour as possible, and in a breadth-first manner
- both $t_{|i}\sigma$ and $t_{|i}\sigma'$ have already been observed: $positive(t_{|i}\sigma) \wedge positive(t_{|i}\sigma')$; we want to explore the behaviour they may lead to in an uniform way, so we will choose the least beaten path: considering vertices $v = \xi^i_*(t_{|i}\sigma)$ and $v' = \xi^i_*(t_{|i}\sigma')$, we will enable event σ if $\nu(v) \leq \nu(v')$ and σ' otherwise.

Let us suppose we find ourselves in the first case, we try to enable event σ, and component C_i does not accept the forwarded message. We assume that, in this case, no transition has been taken and C_i hasn't changed its state. Therefore, we can continue from that point on by trying to enable event σ', or other such controllable events, if their corresponding messages are available. As trace $t_{|i}\sigma$ could not be enforced on C_i, this will be marked in T_i by modifying the transition function from vertex $v = \xi^i_*(t_{|i})$, so that, for event σ, it leads towards the empty vertex: $\xi^i(v, \sigma) = \epsilon$.

If from a control point q in Φ none of the forwarded messages are accepted, or there are no more such messages available, then local component C_i and specification automaton Φ cannot be controllably advanced together.

Then, as earlier mentioned, we try to avoid this situation by simulating interactions in the context E_i. The modelled safe environment $E_i \| \Phi$ is explored by a bounded depth-first search phase, that tries to advance property automaton Φ to some future state from which a different set of controllable events are allowed for local component C_i. If such a future state is found in the safe context, we can forward to C_i other messages than we were previously allowed to, and thus determine a progress in the current execution. If still none of these messages are available for forwarding, or accepted by the local component, and if also within a time bound χ no message is sent by C_i, then we have reached a deadlock and the current run stops.

Assume now that from the current state q of property Φ no transition exists on a controllable event σ'', which is also in the event set of C_i, $\sigma'' \in \Sigma^i$. This means

that, after any trace t' with the same projection $t_{|i}$ on Σ^i as t, we will never try event σ'', so it doesn't matter if C_i can accept $t_{|i}\sigma''$ or not, and testing this would be useless. Therefore, we can mark trace $t_{|i}\sigma''$ as failed in T_i, or $negative(t_{|i}\sigma'')$ and modify the transition function in the same way as if we had tried $t_{|i}\sigma''$ unsuccessfully: $\xi^i(\xi^i_*(t_{|i}), \sigma'') = \epsilon$.

The local behaviour exploration process ends for the black box component C_i when either the maximum number γ of executions has been reached, or all the behaviour of C_i that conforms to Φ has been completely explored at runtime up to depth $m+1$, i.e. its associated trace tree T_i is complete to depth $m+1$, as defined in section 4.3.

Finally, an adaptor for the system is computed by using the classic result of Ramadge and Wonham [67] for controller synthesis in order to obtain a controller for the system plant, and then deriving the adaptor from this controller.

4.5.4 Distributed Exploration Algorithm

Below we give the algorithm for the distributed behaviour exploration process. Basically, localizing the inference process represents a specialization of the original algorithm presented at section 4.3. When the learning process is local, instead of global, only one black box component is considered and interacted with, while the interaction with the other components is simulated by a bounded model checking phase performed on their models.

In order for the local proactive adaptor to interact with, and control the black box component by message forwarding, a set of available messages is considered. This set of messages contains the messages received by the local adaptor from the rest of the local adaptors, that explore the behaviour of the other components in the system. So, basically, the messages used locally by the proactive adaptor, to enable controllable transitions in its corresponding component, are messages sent by the other remote components, intercepted by their local proactive adaptors, and the forwarded .

The **localStep** procedure describes a local behaviour exploration step. The local adaptor selects those available messages that could trigger correct transitions. By correct transitions we understand transitions that conform to the desired temporal property Φ.

Algorithm 8 : Local controllable step

1: **procedure** localStep(C_i) {control local step}
2: $v = nodeAfterTrace(T_i, t_i)$ {current vertex in T_i}
3: $toTry = availableMoves(v, i) \cap correctMoves(q^{\Phi}, \Phi)$
4: $progress = false$
5: **repeat**
6: $leastTaken = \{\sigma' \in toTry \mid minim(\nu(\xi^i_*(t_i\sigma')))\}$
7: $\sigma = pickRandom(leastTaken)$
8: $enable(\sigma, C_i)$ {try message}
9: **if** ack_i **then**
10: $advance(\Phi, \sigma)$ {advance property}
11: $putTrace(T_i, t\sigma)$ {positive sample}
12: $t_i = t_i\sigma$
13: $progress = true$
14: **else**
15: $cutTrace(T_i, t_i\sigma)$ {negative sample}
16: **until** $progress$

The local proactive adaptor tries to forward one of these messages to C_i and observes how the component C_i reacts. If the message is accepted (ack_i), the the property automaton is advanced (by $advance$), the receive event is concatenated to the current execution trace and progress is marked. The current execution trace is then kept as a positive sample in the trace tree (by $putTrace$). However, if the message is not accepted, the potential trace is marked as a negative sample (by $cutTrace$), and another available message is forwarded, etc., until we have tried on all the messages selected in the $toTry$ set.

However, the black box component is not explored in isolation, but within the context represented by the other components and their behaviour. Since the behaviour exploration is local, the context is here simulated by bounded model checking steps taken on the component models. Procedure **localStepWithSimulatedMoves** describes how local steps taken at runtime by the real black box component are integrated with simulated steps on the other component models. It is invoked when a simple local step could not be performed. In this case, we try to advance the property together with the remote component models, to see whether from its next state the property will allow us to forward other messages to the local black box. Before each such step, we store the current state of the system, by $saveGlobalState$. Then, we choose one of the possible

Algorithm 9 : Exploring real and simulated behaviour together

```
1: procedure localStepWithSimulatedMoves(depth) {explore models}
2: if depth > maxDepth then
3:   return
4:
5: for all j = 0, n − 1 do
6:   toTry = (availableMsg(j) ∪ sentMsgBy(j)) ∩ correctMoves(U_j, Φ)
7:   for all σ = msg? ∈ Σ?^i(q) ∧ msg ∈ toTry do
8:     saveGlobalState
9:     advanceSynch(U_j, Φ, σ) {advance property and model}
10:    localStep(C_i)
11:    if progress then
12:      return
13:    else
14:      localStepWithSimulatedMoves(depth + 1)
15:      if progress then
16:        return
17:      else
18:        restoreSavedGlobalState
```

next steps to be taken by a local model U_j of a remote component C_j, and we advance U_j and the property automaton Φ together by $advanceSynch(U_j, \Phi, \sigma)$. After this, we try again to advance the black box, by invoking **localStep**. If we succeed, the procedure returns. Otherwise we perform a recursive call and continue the simulated execution trace until we either succeed to advance the black box, of we have reached the maximum path depth. If the recursive call returns and no progress was made, the previous global state is restored, and we then try to continue by advancing the next model.

Procedure **remoteReceive** describes how incoming messages from other components are received (asynchronously, in an event-based way) and also stored, if they are needed to explore the local component.

Incoming messages from the local component are asynchronously intercepted by procedure **localReceive** – almost similar to the onReceive procedure described at subsection 4.3.1. The intercepted message is stored for later use – by operation $store$. Then, uncontrollable event σ is appended to trace t_i, $putTrace$ adds observed trace $t_i\sigma$ as a positive sample in the trace tree T_i. If the the property model can accept the observed send event σ progress is marked and the

Algorithm 10 Handling asynchronous receives

1: **procedure** remoteReceive(σ, i) {remote message receive listener}
2: $msg = \sigma.message$
3: **if** $\sigma \in \Sigma^i$ **then**
4: $store(msg)$
5:
6: **procedure** localReceive(σ, i) {local message receive listener}
7: $msg = \sigma.message$
8: $store(msg)$
9: $putTrace(T_i, t\sigma)$ {positive sample}
10: $t_i = t_i\sigma$
11: **if** $advance(\Phi, \sigma)$ **then**
12: $progress = true$
13: **else**
14: $stopCurrentExecution = true$
15: $broadcast(msg)$

exploration continues. If not, then we have reached an uncontrollable erroneous transition, and therefore the current execution will end. Received messages are forwarded, by operation $broadcast$, to the other remote proactive adaptors in the system.

Procedure **execution** describes how a local execution for behaviour exploration takes place. The local component is assumed to be component C_i. In the beginning, C_i is reset to its initial state, together with the property automaton and all the models. Then, the following actions are repeated until either the property is violated at runtime by means of an uncontrollable event or the maximum number of execution steps is reached. An attempt to advance the local component is made by invoking first the **localStep** procedure. If it fails, **localStepWithSimulatedMoves** is invoked. When no progress is made whatsoever, the execution stops.

Finally, the behaviour exploration main for local exploration goes as follows. The local trace tree T_i is initialized. The models for the components are either precise, if known, or are assumed to be most general otherwise. Then, the **execution** procedure is repeatedly invoked, until either the trace tree T_i is completely explored up to depth $m + 1$, or the maximum number of executions is reached and the learning process stops.

Algorithm 11 Local controllable execution

1: **procedure** execution {control an execution}
2: reset C_i
3: reset Φ
4: **for all** $j = \overline{0, n-1}$ **do**
5: reset U_j
6:
7: $count = 0$
8: **repeat**
9: $progress = false$
10: localStep(C_i)
11: **if** not($progress$) **then**
12: localStepWithSimulatedMoves(0)
13: **else**
14: $count = count + 1$
15: **until** $\neg progress \ \vee \ stopCurrentExecution$)
16:
17: {behaviour exploration main}
18: initialize(T_i)
19: $execNr = 0$
20: $maxExLength = 2m$
21: **repeat**
22: execution($maxExLength$)
23: $execNr = execNr + 1$
24: **until** $(execNr \geq maxExecNr) \vee (\forall i \leq n-1.$ completelyExplored(T_i))

4.6 Exploration Optimization

4.6.1 The Issue of Cycle Identification

If the maximum allowed number of executions γ of is reached before having all black box components in the system completely explored, the resulting models will represent a less precise approximation. We have already mentioned the importance of identifying in the unknown component behaviour cycles that represent projections, on the component event set, of the cycles in the specification automaton: it makes these cyclic scenarios usable as specified in the composed system. Otherwise, if a cyclic scenario is not identified as such, resetting the system, although undesirable, cannot be avoided when the user aims at repeating the same scenario several times.

Therefore, when learning behavioural models for the black boxes in the system, exploring the cycles in the specification automaton should be prioritized over the rest of all possible behaviours.

Further on, we shall see how this prioritization takes place. Let us denote the simple cycles in the specification Φ by sc^0, sc^1, etc., and let $sc^j_{|i}$ be the cycle resulted from the projection of sc^j on the event set Σ^i. For a cycle sc^j in Φ to also exist in the system plant, it is necessary for all its projection cycles $sc^j_{|i}$ to be found in the behaviour of their corresponding black box components C_i.

Let us now assume that for every component C_i we unroll the projection cycles $sc^j_{|i}$ of the cycles in the specification automaton Φ, and we unroll each of these cycles, separately, as many times as needed to obtain the shortest possible trace of at least $2m$ events. We denote the trace of at least $2m$ events, obtained from unrolling projection cycle $sc^j_{|i}$ as tc^j_i. Also, since a loop can usually be unrolled in more than one way, depending on the starting point, we denote a trace containing only one unrolling of projection cycle $sc^j_{|i}$ as $c^j_{|i}[q]$, where q is the state in Φ starting from which the cycle is unrolled. In order to obtain trace tc^j_i, the cycle is unrolled, before projection, from its entry state q_{s0} in Φ that is closest to the initial state q_0 of Φ.

One special situation is the case when two or more cycles of the specification automaton Φ have a join point, i.e. a common state q_{jp}. Establishing whether the system can pass from one cycle to another during execution is important, since it can ensure alternating various usage scenarios during the same execution, without the need of resetting the system.

Identifying such a join point requires more than individually identifying the cycles. Assuming two cycles, $sc^0_{|i}$ and $sc^1_{|i}$, the traces that have to be explored to determine whether they have a join point are as follows. For simplicity, $c^j_{|i}[q_{s0}]$ will be denoted as $c^j_{|i}$, and $c^j_{|i}[q_{jp}]$, the projected unrolling of the cycle $sc^j_{|i}$ from the join point q_{jp}, is marked as $c'^j_{|i}$. The junction trace between $c^j_{|i}$ and $c'^j_{|i}$, a subsequence of $c^j_{|i}$ is denoted as jt^j_i, and it represents the portion of the unrolling trace of cycle $sc^j_{|i}$ that goes from state q_{s0} to state q_{jp}. Such a junction trace is important, as the execution always traverses this subsequence when passing from the entry point of the cycle to the join point, from where another cycle can be entered.

In order to identify a join point of the two projection cycles $sc^0_{|i}$ and $sc^1_{|i}$, we have to explore those traces that define first the two cycles, and then their connection, the join point. Thus, the following traces need to be explored at runtime:

- traces tc^0_i and tc^1_i must be enabled at runtime in order to first explore the two cycles $sc^0_{|i}$ and $sc^1_{|i}$
- all traces of the form $(c^0_{|i})^k jt^0_i (c'^1_{|i})^{k'}$, must be explored at runtime, which implies exploring the repetition of $c^0_{|i}$ for k times, followed by the repetition of $c^1_{|i}$ for k' times. Here, k varies increasingly from 1 to $\frac{2m}{|c^0_{|i}|}$, while k' decreases from $\frac{2m}{|c'^1_{|i}|}$, to 1, so the resulting trace is the shortest possible of this form that has a length of at least $2m$ events. The last condition can be expressed as $k|c^0_{|i}| + |jt^0_i| + k'|c'^1_{|i}| \geq 2m$, but $(k-1)|c^0_{|i}| + |jt^0_i| + k'|c'^1_{|i}| < 2m$ and $k|c^0_{|i}| + |jt^0_i| + (k'-1)|c'^1_{|i}| < 2m$ – so if one of the cycles is unrolled one less time, the length of the resulting trace is less than $2m$.
- also, all traces of the form $(c^1_{|i})^{k'} jt^1_i (c'^0_{|i})^k$ have to be explored, which means exploring the repetition of $c^1_{|i}$ for k' times, and the repetition of $c'^0_{|i}$ for k times. Here it is k' that varies increasingly from 1 to $\frac{2m}{|c^1_{|i}|}$, while k decreases from $\frac{2m}{|c'^0_{|i}|}$, to 1. The resulting trace is the shortest possible of this form that has at least $2m$ events, so the following condition is satisfied: $k|c'^0_{|i}| + |jt^1_i| + k'|c^1_{|i}| \geq 2m$, but $(k-1)|c'^0_{|i}| + |jt^1_i| + k'|c^1_{|i}| < 2m$ and $k|c'^0_{|i}| + |jt^1_i| + (k'-1)|c^1_{|i}| < 2m$ – just as in the previous case, unrolling any of the cycles one less time makes the trace shorter than $2m$.

Now, let us assume we have p cycles, that share the same join point – the general case. In this case, we will take each pair of different cycle projections $sc^l_{|i}$ and $sc^h_{|i}$ (where $l \neq h$ and $l, h = \overline{0, p-1}$) separately and treat them like we did with cycle projections $sc^0_{|i}$ and $sc^1_{|i}$ presented above. Thus, for each such pair of projection cycles $sc^l_{|i}$ and $sc^h_{|i}$, the traces considered for exploration are tc^l_i, tc^h_i and all traces of the form $(c^l_{|i})^k jt^l_i (c'^h_{|i})^{k'}$ and $(c^h_{|i})^{k'} jt^h_i (c'^l_{|i})^k$, where k and k' are both varying from 1 to $\frac{2m}{|c'^l_{|i}|}$, respectively $\frac{2m}{|c'^h_{|i}|}$, in the same way as described above.

4.6.2 Cycle-Oriented Behaviour Exploration

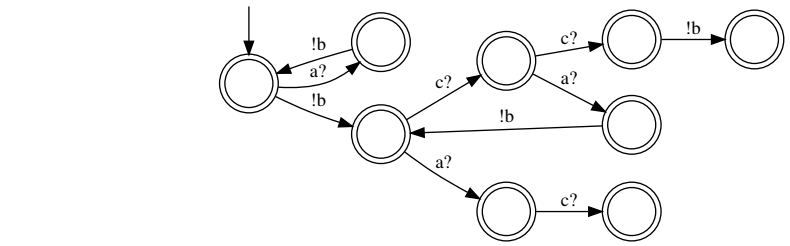
All event sequences considered necessary for cycle-based behaviour exploration are kept in a dedicated trace tree TC_i, one for each black box component C_i. This implies all unrolled cycle traces tc^j_i for component C_i, representing the unrolled projection cycles for all cycles in Φ, and also includes the particular unrolled traces needed to properly identify those cycles that have a common join point. Further on, we use the trace tree TC_i to decide on which controllable event to enable for black box component C_i, when guiding a current execution.

Let the transition function for TC_i be denoted as $\kappa^i : VC^i \times \Sigma^i \to VC^i$, where VC^i is the set of vertices for trace tree TC_i. The extension of the transition function to strings of events will then be denoted as $\kappa^i_* : \Sigma^i_* \to VC^i$.

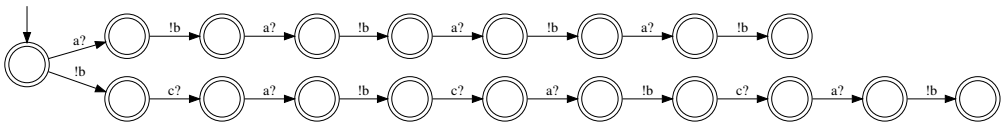
Let us see now how the exploration process takes place when the exploration process is optimized for cycle discovery. This means that its priority is to first explore the cycles in the specification automaton Φ. As before, assume the current execution trace is denoted by t, with $t_{|i}$ the projection of trace t on the event set Σ^i, corresponding to component C_i.

Assume also that $t_{|i}$, the projection on Σ^i of the current execution trace, exists in the TC_i as a prefix of an unrolled cycle projection tc^j_i, and so $\kappa^i_*(t_{|i}) = v$. This means that we are currently exploring one of the relevant cycles of specification Φ, as it is projected on the alphabet of component C_i. As we want the ongoing execution to be meaningful for our purpose – identifying the relevant cycles – we will choose to enable a controllable event that either keeps the execution on the current cycle, or exploits a join point to switch to another relevant cycle.

Then, suppose that the transition function of TC_i, $\kappa^i(v, \sigma)$, is defined only for



(a)



(b)

Figure 4.9: A safety property – 4.9(a), and the dedicated trace tree TC that contains its relevant cycles – 4.9(b)

controllable event $\sigma' \in \Sigma^i$. This means that the only way to keep the current execution on a relevant cycle is to enable controllable event σ'. Then, σ' will be the next event enabled in component C_i, if possible, as continuing with $t_{|i}\sigma'$ in C_i is the key to continue exploring the current projection cycle in C_i. If for several controllable events $\sigma' \in \Sigma^i$, the transition function $\kappa^i(v, \sigma')$ is defined, this means that there are several possibilities to guide the current execution on a relevant cycle. In this case, these events will be enabled on a least-explored first basis. If, however, the set of controllable events, for which transition function $\kappa^i(v, \sigma')$ is defined, is currently empty, then the current execution will continue by enabling any controllable event allowed by Φ, as usually done during the unoptimized behavioural exploration.

For example, assume we have one black box component, on which we want to impose the desired safety property in figure 4.9(a). The relevant cycles in the property automaton are kept in the tree in figure 4.9(b). Then, assume we are during the exploration process of the black box component, and that the current execution trace is $b!c?$. The controllable choices allowed by the property automaton, at this point, are to enable either $a?$ or $c?$ for execution, and in an unoptimized exploration process we would choose the event that has been enabled the least so far. However, in the cycle-optimized exploration process,

the chosen option is the one that allows us to explore a relevent cycle, so event $a?$, the one that follows the prefix $b!c?$ in the tree at figure 4.9(b), is the one to be enabled.

The cycle-oriented exploration process continues for component C_i until for each prefix trace $tc' \in TC_i$ such that $|tc'| \leq m+1$, if trace tc' is an access trace for a vertex v' in T_i, then vertex v' is completely explored. This condition is important, as it allows for distinct vertices with a difference in depth of at most m to be merged into the same state (thus obtaining cycles) when building the learned model.

$$\forall tc' \in TC_i \; s.t. \; \xi^i_*(tc') = v' \wedge v' \in V^i \setminus \epsilon \rightarrow complete(v')$$

After the condition above is met for a component C_i, the behavioural exploration of C_i will continue as described in the previous section, by uniformly exploring all potential execution traces allowed by safety property Φ, until component C_i is completely explored to depth $m+1$.

4.7 Adaptor synthesis

The final aim of the technique is to correctly compose the system S from the components $C_i, i \leq n-1$. This takes place by computing a centralized adaptor for the system S. No matter of the learning setting used – centralized, distributed, or even individual – the solution we have adopted for system composition relies on centralized adaptation, as it is a simple and general approach, applying to all safety properties.

The centralized adaptor for the system is computed starting from a centralized controller $Ctrl$ that enforces the property Φ over the system plant. The controller $Ctrl$ is obtained using the elements of control theory described in section 2.5, and specifically the fixpoint procedure for controller synthesis.

Consider a learned model of a component C_i to be denoted by U_i. Also, let $U_\times$ be the asynchronous product of all the learned models: $U_\times = U_0 \times U_1 \times ... \times U_{n-1}$. Thus, $U_\times$ models the system plant, on which the specification Φ is to be enforced. The computation of the controller, noted by $Ctrl$, for the specification automaton Φ, relies on the classical result of Ramadge and Wonham: $Ctrl = \mathbf{supcon}(U_\times, \Phi)$, where **supcon**, described in [67], and also

detailed in section 2.5, is the mentioned fixpoint procedure. All behaviours of the plant $U_\times$ that do not violate the safety property Φ are allowed by the computed controller $Ctrl$.

Finally, adaptor A is obtained from the computed controller $Ctrl$ by mirroring its event set. This means that each transition in $Ctrl$ triggered by a message receive event $\sigma = msg?$ will be triggered in A by its corresponding message send event $\sigma' = msg!$, and conversely, each transition triggered in $Ctrl$ by a message send event will become in A a transition on the corresponding message receive event.

This mirroring is necessary due to the subtle difference between an adaptor and a generic controller: the adaptor manifests its control on the system plant by forwarding or consuming messages. Thus, an adaptor enables a controllable, message receive event by forwarding the expected message, which triggers the transition associated with that receive event in the receiving component. Forwarding specifically means that the adaptor has to send the expected message to the component, so in the adaptor FSM, the transition to be taken will be labeled by a message send event. Also, the adaptor has to react to any uncontrollable event in the plant, i.e. to any message send event, by receiving the message sent by the component, so in the adaptor FSM, the transition to be taken in this case will be labeled by a message receive event. By contrast, a generic controller, as computed by the Supremica tool [59], has transitions on exactly the same uncontrollable events that occur in the plant, and on the same controllable events to be enabled in the plant. This happens because the generic controller regards controllable and uncontrollable events not as message receive, respectively message send events, but in a more general sense.

4.8 Proofs

In this section, we prove several properties of our algorithm, and of the learned models, properties that are vital for the validity of our approach.

We start by proving that the assumed bound m, the length of longest acyclic path in the model, is a valid bound for path exploration.

Then, we prove that building the model from the trace tree associated with the black box component is permissive with respect to the observed behaviour of the black box. We also prove that the models learned by our algorithm are safe approximations of the real component behaviour, and therefore the final obtained adaptor is also safe for the real system.

Finally, we prove that, assuming we can run it for an unlimited time, the model learning algorithm eventually reaches its termination condition.

4.8.1 State reachability

Theorem 1. *In any finite state machine M, any state can be reached from the initial state by a path of at most m transitions, where m is the length of the longest acyclic path in the model.*

$$\forall q \in Q^M, \exists \in \Sigma_*^M. (\delta_*^M(t) = q) \rightarrow |t| \leq m \tag{4.1}$$

Proof. Assume that for a finite state machine M, a state q exists so that q cannot be reached by a path of m transitions or less.

$$\exists q \in Q^M, \forall t \in \Sigma_*^M. (\delta_*^M(t) = q) \rightarrow |t| > m \tag{4.2}$$

***Case* 1.1.** *Assume t s.t. $\delta_*^M(t) = q$ contains no cycles.*

Then, from 4.2:

t acyclic $\wedge$ $|t| > m \rightarrow m$ is not the longest acyclic path $\rightarrow$ contradiction

***Case* 1.2.** *Assume $t = uc^p w$, where c is a cycle in the model, while u and w are acyclic sequences.*

Then $\delta_*^M(t) = q \rightarrow \delta_*^M(uw) = q$, *where sequence* uw *is acyclic.*

$|uw| \leq m \wedge$ *4.2* $\rightarrow$ *contradiction*

$|uw| > m \rightarrow$ *same as case 1.1*

Therefore, assumption 4.2 is false and any state in the model is reachable by a trace of at most m transitions.

□

4.8.2 Permissiveness

Definition 1. *A finite state machine* M *over a set of events* Σ *is permissive with respect to a set* ES *of sequences over events in* Σ, $ES \subseteq \Sigma_*$, *when* M *accepts all sequences in* ES.

Theorem 2. *The deterministic prefix-closed model* U, *built from trace tree* T *by the model building algorithm, is permissive with respect to the set of observed event sequences, i.e. accepts any observed sequence.*

$$\forall t \in \Sigma_*.\, positive(t) \rightarrow t \in \mathcal{L}(U) \tag{4.3}$$

Proof. Suppose 4.3 is false and the learned model is not permissive with respect to the observed sequences. Then:

$$\exists t \in \Sigma_*.\, positive(t) \wedge t \notin \mathcal{L}(U) \tag{4.4}$$

Without loss of generality, assume $t = uw$, with both $u, w \in \Sigma_*$, and $u \in \mathcal{L}(U)$.

Assume also that after sequence u, model U is in state q: $q = \delta_*(u)$.

Also, in trace tree T, sequence u is the access trace of a unique vertex v: $v = \xi_*(u)$.

$$(positive(t) \wedge t = uw) \rightarrow \xi_*(uw) \in V \tag{4.5}$$

From 4.5 we have

$$(positive(t) \wedge t = uw) \rightarrow w \in \mathcal{L}_{\surd}(v) \tag{4.6}$$

Now, from $q = \delta_*(u)$ in built model U and $v = \xi_*(u)$ in trace tree T, and due to the fact that both structures U and T are deterministic, we have $q \asymp P$, where P is a partition of vertex set V and $v \in P$.

Denoting $\mathcal{L}(U, q)$ as the language of state q in the built model U, 4.4 implies that

$$\exists t \in \Sigma_*.\, positive(t) \wedge t = uw \wedge q = \delta_*(u) \wedge w \notin \mathcal{L}(U, q) \tag{4.7}$$

Assuming that, for states $q, q' \in Q$ and partitions P, P' over vertex set V we have $q \asymp P$ and $q' \asymp P$, the model building algorithm ensures that

$$\forall v \in P, \forall \sigma \in \Sigma.\, \xi(v, \sigma) = v' \wedge v' \in P' \rightarrow \delta(q, \sigma) = q' \tag{4.8}$$

By transitive closure, this means that for any state $q \in Q$, its language is the reunion of observed languages for all vertices v in partition P, where $q \asymp P$:

$$\forall q \in Q.\, q \asymp P \rightarrow \mathcal{L}(U, q) = \bigcup_{\forall v \in P} \mathcal{L}_{\surd}(v) \tag{4.9}$$

But from 4.6 and 4.7 we have that $\exists w \in \Sigma_*$ s.t. $w \in \mathcal{L}_{\surd}(v)$ and $w \notin \mathcal{L}(U, q)$, where $q \asymp P$ and $v \in P$. Therefore, contradiction with 4.9.

Thus, the built model U is permissive with respect to the set of all observed event sequences. □

4.8.3 Safety

Definition 2. *A controller $Ctrl$ that enforces a temporal property Φ over a plant $\mathcal{P}$ is safe if all possible behaviours of controlled plant $Ctrl \| \mathcal{P}$ satisfy Φ:*

$$\mathcal{L}(Ctrl \| \mathcal{P}) \subseteq \mathcal{L}(\Phi) \tag{4.10}$$

Definition 3. *An adaptor A is safe for a system $\mathcal{S}$ and a property Φ iff its corresponding controller $Ctrl$ is safe for the plant $\mathcal{P}$ represented by $\mathcal{S}$, and the temporal property Φ.*

Theorem 3. *Adaptor A, obtained for learned models U_i and property Φ, is safe.*

Proof. If precise models U_i were learned, the proof is trivial, so we only treat below the case of learning approximate models U_i.

Suppose that adaptor A is not safe. Then, the controller $Ctrl$ over plant $\mathcal{P} = U_\times$ is also not safe. Thus:

$$\exists t \in \Sigma_*.\, t \in \mathcal{L}(Ctrl \| R_\times) \wedge t \notin \mathcal{L}(\Phi \| R_\times) \longrightarrow t \in \mathcal{L}(R_\times) \setminus \mathcal{L}(\Phi) \tag{4.11}$$

$$Ctrl = \mathbf{supcon}(U_\times, \Phi) \longrightarrow \mathcal{L}(Ctrl || U_\times) \subseteq \mathcal{L}(\Phi) \tag{4.12}$$

$$(4.11) \wedge (4.12) \longrightarrow t \in \mathcal{L}(Ctrl \| (R_\times \setminus U_\times)) \tag{4.13}$$

Let $t = u\sigma v$, where $u \in \mathcal{L}(Ctrl||U_\times)$ is a correct prefix, σ is the error-inducing event, i.e. the first event of an event suffix unspecified by Φ, and v represents the rest of the trace. Let $\sigma \in \Sigma^i$. Then:

$$(4.13) \wedge u \in \mathcal{L}(Ctrl||U_\times) \longrightarrow u\sigma \notin \mathcal{L}(Ctrl \| U_\times) \tag{4.14}$$

***Case* 3.1.** *Assume $\sigma \in \Sigma^i_?$.*

But this means σ controllable. Then $\sigma \in \Sigma^i_? \wedge u\sigma \notin \mathcal{L}(\Phi) \rightarrow u\sigma \notin \mathcal{L}(Ctrl) \rightarrow u\sigma \notin \mathcal{L}(Ctrl \| R_\times) \rightarrow$ contradiction.

***Case* 3.2.** *Assume $\sigma \in \Sigma^i_!$.*

$$(4.14) \longrightarrow \sigma \in \Sigma^i_! \wedge u_{|i}\sigma \notin \mathcal{L}(U_i) \tag{4.15}$$

Each state $q_j \in Q^i$ corresponds to a finite set of vertices $P_j = \{v_{k_0}, v_{k_1}, ..., v_{k_j}\}$, resulted from compatibility partitioning over the vertex set V^i of trace tree T_i.

If $\sigma \notin \Sigma^i(q)$, and q corresponds to the compatibility partition P, $q \asymp P$ this means that:

$$\sigma \notin \Sigma^i(q) \wedge q \asymp P \rightarrow \forall v \in P.\, \xi^i(v, \sigma) = \epsilon \tag{4.16}$$

Thus, 4.16 implies that the transition on σ was incorrectly removed from each and every vertex $v \in P$.

Then, from 4.16 we have $\xi^i(v, \sigma) = \epsilon, \forall v \in P$.

Since $\delta^i_*(u_{|i}) = q$ and $q \asymp P$, there exists a vertex $v' \in P$ s.t. $v' = \xi^i_*(u_{|i})$.

$$\exists v' \in P, v' = \xi^i_*(u_{|i}) \wedge \xi^i(v', \sigma) = \epsilon \tag{4.17}$$

Since C_i is fair by bound θ:

$$\sigma \in \Sigma^i_!(q) \wedge \nu(\xi^i_*(u_{|i})) \geq \theta \wedge \neg obs(u_{|i}\sigma) \longrightarrow \xi^i_*(u_{|i}\sigma) = \epsilon \tag{4.18}$$

Also, from the model building phase:

$$\sigma \in \Sigma^i_!(q) \wedge \nu(\xi^i_*(u_{|i})) < \theta \wedge \neg obs(u_{|i}\sigma) \longrightarrow \xi^i_*(u_{|i}\sigma) = \zeta \tag{4.19}$$

Thus, from 4.17, 4.18 and 4.19, it results that

$\exists v' \in P, v' = \xi^i_*(u_{|i}) \wedge \xi^i(v', \sigma) = \epsilon \rightarrow \nu(v') \geq \theta \wedge \neg obs(u_{|i}\sigma)$

But, if $\nu(u_{|i}) \geq \theta \wedge \neg obs(u_{|i}\sigma)$, then $u_{|i}\sigma$ was dismissed as infeasible based on the fairness assumption.

Since $u_{|i}\sigma$ was observed at runtime, the fairness assumption is false $\rightarrow$ contradiction.

From $Case\ 3.1$ and $Case\ 3.2$, we have that the feasible partial trace $u_{|i}\sigma \in \Sigma^i_*$, cannot be incorrectly cut out from trace tree T_i, and thus incorrectly left out from learned model U_i. Therefore, the obtained system controller $Ctrl$ is safe, and, thus also adaptor A is safe. □

4.8.4 Termination

Theorem 4. *The learning process eventually terminates.*

Proof. If a bounded number of executions is considered and approximate models are learned, the proof is trivial.

Suppose we try to learn precise models. The learning ends when the trace tree T_i is complete up to depth $m + 1$, which is achieved in at most θ^{m+1} queries, due to the fairness condition. The consistency check ends, worst case, when trace tree T_i is complete up to depth $2m$, in at most θ^{2m} queries. Therefore, the algorithm always terminates.

□

4.9 Complexity Limitations. Discussion

4.9.1 Asynchronous Angluin

The total number of queries needed by the Angluin algorithm to learn a deterministic finite automaton [4] is:

$$\mathcal{O}(|\Sigma|ln^2)$$

where:

- Σ is a set of input-enabled (and thus controllable) events
- n is the number of states of the minimum DFA
- l is the maximum length of a counterexample

To the membership queries above, one has to further add the potential cost of equivalence queries needed. When learning the behaviour of a black box component the ideal equivalence oracle assumed by the Angluin algorithm [4] doesn't exist. Instead, it is usually implemented by the Vasilevskii-Chow conformance testing algorithm [79], which has a complexity of

$$\mathcal{O}(l^2n|\Sigma|^{n-l+1})$$

This leads to a learning complexity of

$$\mathcal{O}(|\Sigma|ln^2 + l^2n|\Sigma|^{n-l+1})$$

Now, assume that the event set Σ contains both controllable and uncontrollable events: $\Sigma = \Sigma_? \cup \Sigma_!$, and that we have a fairness bound θ, as assumed above, so that if a state q is reached for at least θ times, all uncontrollable events σ that can occur from q were observed at least once.

If we want to try a membership query for a trace t, that contains η uncontrollable events, we will have to query for t up to θ^η times. Since $0 \leq \eta \leq l$, the maximum number of queries needed to try a certain trace t becomes θ^l.

Therefore, the total number of membership queries needed by the Angluin algorithm to learn a complete model for an asynchronous black box is:

$$\mathcal{O}(|\Sigma|ln^2\theta^l + l^2n|\Sigma|^{n-l+1}\theta^l)$$

It is interesting to notice here that repeatedly asking a specific membership query can provide positive answers to other, possibly future queries. This can lead to important savings when all observed queries are kept in memory, thus some queries can be answered instantaneously. However, obtaining negative answers to a future query is more complicated. The emphasis put on the number of queries is justified by the fact that runtime queries are usually the most expensive part of the learning process [55].

It is important to note here that, due to the fact that the Angluin algorithm does not infer safe approximations of the real black box behaviour, the learned model has to be complete before we can safely use it to compose a system. This is not always feasible.

4.9.2 BASYL

Our usual exploration method is breadth-first, and, considering a most permissive property, it needs at most $|\Sigma_?|$ attempts from the same vertex, in order to try all controllable events. Let us introduce α and β so that:

$$\alpha = min(\frac{\theta}{2}, |\Sigma_?|)$$

$$\beta = \frac{\theta}{2} + max(\frac{\theta}{2} + |\Sigma_?|)$$

Thus, β is the upper bound on the number of times a vertex v in the trace tree has to be reached in order to be completely explored, and θ is the fairness bound.

We explore the behaviour of the black box using execution traces of length $2m$, where m is the maximum length of an acyclic event sequence in the real model of the black box.

Let us note by n_t the number of traces needed by BASYL to obtain a precise model. First, the trace tree will have to be completely explored up to depth $m+1$, which will require at least α^{m+1} and at most β^{m+1} traces. Then, exploring also

the found differentiating traces, as earlier described, can lead to a final upper bound of traces needed to obtain a precise model equal to β^{2m}.

We can also derive a lower bound on n_t, assuming no differentiating traces were found, as α^{m+1}. Thus, in the end we have:

$$\alpha^{m+1} \leq n_t \leq \beta^{2m}$$

Considering the number of events n_e required by the learning process, the following bounds result:

$$m\alpha^{m+1} \leq n_e \leq 2m\beta^{2m}$$

Let us now consider the operations during the equivalence partitioning phase of the model building process. Vertices of depth at most $m+1$ are examined and compared each to each for the equivalence check. From each vertex, at most Σ valid branches emerge, thus the total number of vertices on a depth less than, or equal to $m+1$ is $\frac{|\Sigma|^{m+1}-1}{|\Sigma|-1}$.

Thus, the number of comparations performed is $\frac{1}{2}\left(\frac{|\Sigma|^{m+1}-1}{|\Sigma|-1}\right)^2$.

In the equivalence check phase, the traces from each vertex are considered on a depth of at most m, this leading to a maximum of $|\Sigma|^m$ traces explored for one vertex. Finally, this leads to a total number of operations of at most

$$\frac{1}{2}\left(\frac{|\Sigma|^{m+1}-1}{|\Sigma|-1}\right)^2 |\Sigma|^m$$

Once the equivalence partitions are established, building the automaton requires a traversal of the tree up to a depth of m, thus the final phase requires a number of $|\Sigma|^m$ operations.

Therefore, the complexity of the model building process is
$\mathcal{O}(\frac{1}{2}\left(\frac{|\Sigma|^{m+1}-1}{|\Sigma|-1}\right)^2 |\Sigma|^m + |\Sigma|^m) = \mathcal{O}(\left(\frac{|\Sigma|^{m+1}-1}{|\Sigma|-1}\right)^2 |\Sigma|^m) = \mathcal{O}(|\Sigma|^{3m})$

4.9.3 Discussion

In the case of learning complete models, an upper bound on the number of queries required to learn a partially controllable black box using the Angluin algorithm is $|\Sigma|ln^2\theta^l + l^2n|\Sigma|^{n-l+1}\theta^l$, while for BASYL is β^{2m}. Assuming that $l = m$, the two methods have similar complexities for $n \geq 2m$ – when the maximum number of states considered is at least twice the assumed size of the longest acyclic path in the model. Otherwise, if $n < 2m$, thus for "deep" behavioural models, the considered adaptation of the Angluin algorithm is more efficient.

If, however, because of a limited amount of available queries, the Angluin algorithm fails to provide a confirmed conjecture, its latest model hypothesis might not be a safe approximation. This is easily provable since, in the absence of a successful equivalence query, an uncontrollable counterexample is an undismissable possibility. The BASYL approach, however, is specifically designed to employ any available number of queries and to obtain a safe approximation of the real black box behaviour.

When a safety property Φ is considered, the learning is further reduced to traces allowed by Φ, so in this case the behavioural model learned is not the real model R of the black box behaviour, but $R\|\Phi$, its synchronous product with the safety property. We did not, however, considered this case separately while discussing the complexity of the learning process, since learning $R\|\Phi$ is similar to learning R, with eventual speed-ups resulting from specific properties of $R\|\Phi$.

Chapter 5

Experimental Results

5.1 Tool Support: BASYL

In order to validate the theoretical approach described in this work, we have implemented our proactive model inference solution as the BASYL (Black-box ASYnchronous Learning) prototype. All performed experiments were run on an Acer Aspire 3820TG with a 2.26 GHz Intel Core i5 430M processor, a 4 GB RAM memory and a 3 MB cache.

BASYL is entirely written in Java, under Java 2 Standard Edition 6.0, and was tested under Linux, Fedora Core 16 distribution. It consists of a total of 149 classes and more than 26000 lines of code. It contains a monitoring component, that observes the events occurring at runtime and advances the models accordingly, a decision logic component that chooses the next controllable transition to be enabled (with special extensions for cycle-oriented exploration and runtime consistency check), specific learning coordinator components for both the centralized and the distributed approach, implementations of the abstract data structures described in this work (trace trees, finite state machines, etc.).

For each black box to be learned, an instance of the monitor is created, that in turn creates an instance of the needed trace tree, and, if necessary, an instance of the cycle trace tree. If the learning is centralized, one inference coordinator and one decision logic component are instantiated for the whole process, otherwise, if the learning is distributed, several coordinators and decision components are created, one for each black box in the system. The results of the inference process are provided as a list of finite state machine objects, that are printed to

file in the XML format used by Supremica [59].

In order to link the high abstraction level at which BASYL works to a concrete technology, technology specific drivers are needed – for some of the case studies presented below, we implemented such a test driver for JMS. However, for simplicity, the results presented in this work only refer to experiments performed on abstract simulations of real systems, without making an actual use of any technology specific test drivers.

The basic abstractions used in the implementation of BASYL are:

- `AbstractMessage`, implemented by `MockMessage`, `JMSMessage`: an object representing a message sent or received by a component; the message is mainly characterized by its name and its index in the system's `AbstractAlphabet`.
- `BehaviourModel`, implemented by `BasicFSM`, the latter extended by `FiniteStateMachine` and `FSMProperty`: the finite state machine used to represent the behaviour of the component.
- `TraceTree`, implemented by `SafeTraceTree` and `WildTraceTree`: the interface for the trace tree structure containing the runtime observation traces. The `WildTraceTree` object is used to mark the mentioned "unknown future" vertices in a `SafeTraceTree` object. A `SafeTraceTree` entity can be used either to keep the positive and negative observed sequences, or as a cycle trace tree, to keep unrolled traces for cyclic scenarios to be explored.
- `AbstractCoordinator`, implemented by `JMSCoordinator` and also by `JMSLocalCoordinator`: the actual test driver interface, where entity `JMSCoordinator` is the driver for the centralized learning setting, while the entity `JMSLocalCoordinator` addresses the distributed one. However, for learning only simulations of real components, an entity of type `MockCoordinator` is used, extended by `MockDriver` for learning a black box in isolation, by `MockCentralDriver` for centralized inference and by `MockLocalDriver` for the distributed learning process.
- `Monitor`, implemented by `ConcreteMonitor` and `ConservativeMonitor`: the monitoring component. `ConcreteMonitor` monitors a black box, reacts when messages sent or received by the black box are intercepted, respectively acknowledged, and keeps track of the current execution for its associated black box. `ConservativeMonitor` does the same, but for

well-specified components.

- `ExploreGuide`, extended by `CycleGuide` and `ConsistChecker`: the component guiding a current execution for specific purposes. An entity of type `CycleGuide` is used to steer the execution towards the exploration of cyclic scenarios, while the `ConsistChecker` forces the consistency check process needed when learning precise models.

During our experiments, when controllers were computed for the considered systems, in order to ease the complexity burden on the controller generation process, we have left the input/output buffers out, thus implicitly considering the case of unbounded buffers. However, this only affects the size of the resulting controller, and does not interfere with the main results – inferring safe approximations for black box components with uncontrollable events.

When experimenting with the cycle-based exploration, as a scenario-oriented optimization for model learning, we feed the tool the specified cycles. The format we use to describe the interesting cycles, and their join points, is specified and exemplified below.

```
<event><event>:<event><event>[join point index]<event>

<event>:[join point index]<event><event>

...
```

A specific send or receive event is described between angle brackets by the name of its sent/received message, and an exclamation mark for sent messages, or a question mark for received messages. The beginning of the cycle is specified by a colon, and the string of events before the colon signify the access trace of the cycle. If several cycles have one or more common join points, the indexes of these join points are specified between square brackets.

The list of specified cycles for each explored black box is kept in a file with the extension ".cyc". An example of such a cycle specification file, for the Media Renderer component of the Domotics case study, is included below:

```
:[1]<pon?><ponok!>[2]<poff?><poffok!>

:[1]<pon?><ponok!>[2]<play?><playok!>[3]<poff?><poffok!>
```

```
:[1]<pon?><ponok!>[2]<play?><playok!>[3]<pause?><pauseok!>
<poff?><poffok!>

<pon?><ponok!>:[2]<play?><playok!>[3]<stop?><stopok!>

<pon?><ponok!>:[2]<play?><playok!>[3]<pause?><pauseok!><stop?>
<stopok!>

<pon?><ponok!><play?><playok!>:[3]<pause?><pauseok!><stop?>
<stopok!>
```

We can see that there are three join points between the specified cycles. The first cycle has a first join point with the following two, right at its entrance. After the first two events take place, it has a second join point, from which the execution can switch into any of the following four cycles. The third join point appears in the second cycle, and links together all cycles but the first.

For a better understanding of cycle specification, we also include its description in the BNF form:

```
<cycle_list> ::= <cycle> <EOL> <cycle_list> | <cycle>

<cycle> ::= <access_event_list> ":" <cycle_event_list>

<access_event_list>::= <event> <access_event_list> | <event>

<cycle_event_list>::= <join_point> <cycle_event_list>
| <event> <cycle_event_list> | <event>

<event>::= "<" <message_name> "!" ">"
| "<" <message_name> "?" ">"

<join_point> ::= "[" <join_point_index> "]"
```

Most of the experiments performed focus on learning safe approximate models of the black box components, which is the main contribution of this work. In the case of components with uncontrollable events, learning a complete model is much harder than in the case of perfectly controllable black boxes, and seldom feasible in practice. Out of the case studies presented here, only for the one of the Single Sign On protocol did precise model inference prove feasible, as the

involved components had small enough alphabets and fairness bounds, and also their maximum acyclic traces were relatively short.

5.2 Case Study: The Single Sign On Protocol

In the following we introduce the case study of the Single Sign On protocol. The Single Sign On protocol enables a user to gain access to a multitude of independent, but related services, usually belonging to the same company, by only logging in once, in the beginning. The version of Single Sign On we study is the SAML Single Sign On (SSO) Service for Google Applications. [36]

The SAML SSO protocol considers three entities: an identity provider, a service provider and a client. In order to gain access to the desired functionality, the client first sends a request to the service provider. If the client is not authenticated, the service provider redirects it towards an identity provider. The client then authenticates via the identity provider, and then, using the identity provided by the identity provider, returns to the service provider and reissues his initial request. The service provider is then able to fulfill his request and grant him access to the desired service.

For our experiments, we have built an abstract, asynchronous version of the SAML SSO protocol.

The set of messages exchanged contains the following:

- req: the request message issued by the client to the service provider
- redirect: used by the service provider to redirect the client to the identity provider
- reqid: used by the client to request the identity provider for authentication
- cid: contains the certificate provided by the identity provider to the client
- id: contains the certificate to be passed to the service provider
- rbad: issued by the service provider in reply to a malformed request
- badid: issued by the service provider in reply to a bad certificate
- ok: used to notify the client that his request was successfully fulfilled

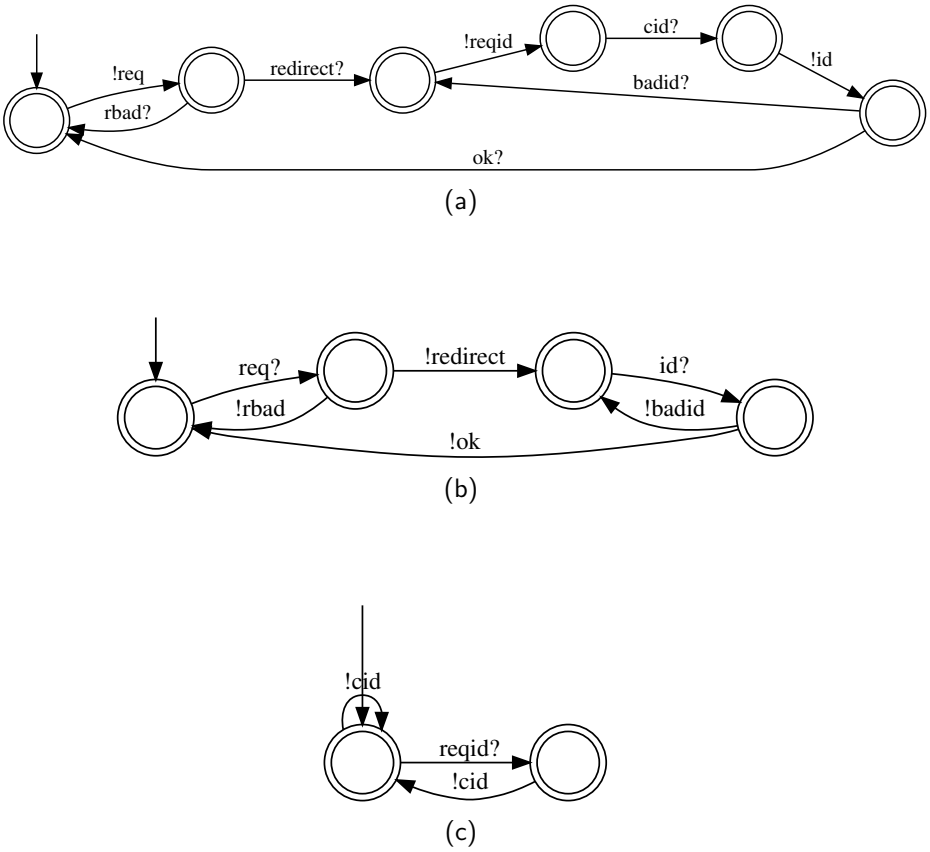
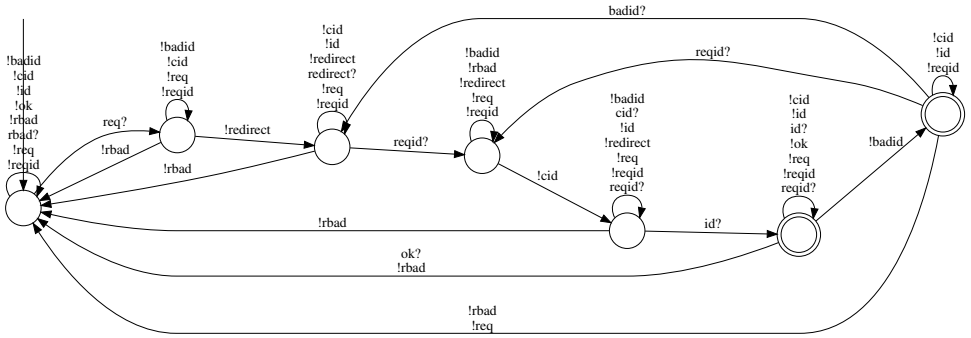


Figure 5.1: (a) SSO client real model, (b) SSO service provider real model, (c) SSO identity provider real model

The SSO client component sends messages *req, reqid, id,* and receives messages *redirect, cid, rbad, badid* and *ok*. The SSO service provider component sends messages *redirect, rbad, badid*, while receiving *req* and *id*, and, finally, the SSO identity provider receives *reqid* and sends *cid*. The real models of the SSO entities can be seen in figure 5.1.

The temporal property we want to enforce on the SSO entities is shown in figure 5.2. The semantics of the desired policy is the following: " the client has to be able to get an identity certificate and use it to address the service provider, and will either have his request accomplished, or his certificate rejected". We have considered all the entities taking part in the SSO protocol as black box components, and attempted to learn their models.



(a)

Figure 5.2: Property to be enforced on the SAML system

In our experiments, we have considered all three components in this case study as black boxes. We have studied the learning process of all the SAML-SSO entities in three situations. First, we learn each entity (the identity provider, the service provider and the client) separately, without considering the safety property. Then, we study the centralized learning process, employing the required safety property in central execution coordination. In the end, we also experiment with the distributed learning setting on this case study, making use again of the safety property at figure 5.2.

The cycle specifications files for all the three components in this case study are included below. They are the same to be used during centralized, distributed and individual behaviour exploration.

The first cycle specification file included describes the expected cycles for the SSO Identity Provider component. The two specified cycles describe the first – the scenario in which after receiving an identity request, the identity provider replies by repeatedly the provided identity, and the second – after the request, the provider replies once, after which continues with processing another request, and so on, and so forth.

```
<reqid?>:<cid!>

:<reqid?><cid!>
```

The second cycle specification file describes the expected cycles for the SSO Service Provider component. The four cycles specified are as follows.

The first one describes that the service receives a request, redirects it towards the identity provider, and then repeatedly receives identity messages, replying with a "bad id" message until an appropriate identity message arrives. The last three cycles all have a common join point in the beginning. Thus, the service provider can repeatedly either receive a request, redirect it, receive a good identity message and comply with the request, or refuse to comply with the request if inappropriate, after receiving a good identity message, or refuse to comply with the request in the first place, instead of redirecting it.

```
<req?><redirect!>:<id?><badid!>

:[1]<req?><redirect!><id?><ok!>

:[1]<req?><redirect!><id?><rbad!>

:[1]<req?><rbad!>
```

Finally, the third cycle specification file contains the expected cycles for the SSO Client component. In the first one, after issuing a request message to the service provider, and receiving its redirect message, the client repeatedly requests an identity certificate from the identity provider, receives it, and sends it to the service provider, while the service provider answers with an identity certificate rejection message.

The last three cycles have a common join point in the beginning. The second cycle goes as follows: the request is issued, the redirecting message received, the identity certificate requested from the identity provider, received, and then sent to the service provider, after which the message corresponding to the successful fulfillment of the request is received. The third cycle is very similar, only that, in the end, a message that signals a bad request is received from the service provider. This message of refusal can, however, be received also before the redirecting message, right after the client initially issues its request, as it is the case for the fourth specified cycle.

```
<req!><redirect?>:<reqid!><cid?><id!><badid?>

:[1]<req!><redirect?><reqid!><cid?><id!><ok?>

:[1]<req!><redirect?><reqid!><cid?><id!><rbad?>

:[1]<req!><rbad?>
```

5.2.1 Individual Exploration

When learning each component in isolation, we have set the parameter m separately for each component, instead of using a global value. Although the use of safety property is not employed, the learning process for incomplete models is studied both with its original, unaltered exploration method and with the cycle-based optimization for behaviour exploration. In the latter case, however, in the absence of the safety property, the cycles used in steering the exploration are the ones used in the centralized and distributed exploration settings.

SSO Identity Provider

The experiments performed to learn in isolation the model of the SSO Identity Provider have used the following values for the relevant parameters: m, the size of longest acyclic trace, was set to $m = 2$, and θ, the fairness bound, was computed as $\theta = 2$.

The average number of executions needed to learn a complete model for the SSO Identity Provider was found as 15.2, out of 10 different, independent learning experiments, which employed the complete model learning algorithm.

In figure 5.3 we can see the results of the approximate model learning algorithm after 5, 10 and 15 executions. It is important to note that for each of the three situations the learning process has been restarted all-over, instead of being resumed from a previous intermediate state for the latter two. This was necessary because the trace tree needs to be preprocessed before equivalence partitioning and actual model building, and this preprocessing makes it useless for further learning, if the learning were to be resumed.

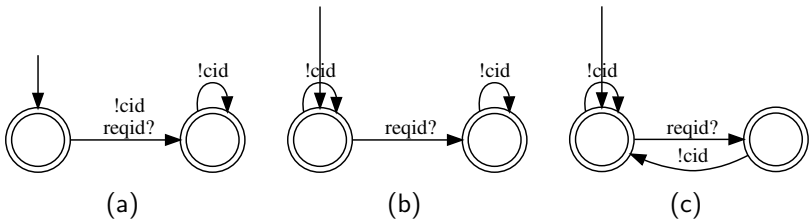


Figure 5.3: Learned models for SSO service provider in: (a) 5 executions, (b) 10 executions (c) 15 executions.

The SSO Identity Provider is a trivial, and thus easy to learn component. After 5 executions, the model obtained in 5.3(a) shows that the transitions from the initial state have been properly identified, but, since some of the vertices in the equivalence partition corresponding to the second state are still insufficiently explored, the second state has a self-loop on the one uncontrollable event in the alphabet, signifying the unknown future. Also, the cycle on event $cid!$ from the initial state has not been identified.

After 10 executions, the cycle from the initial state on event $cid!$ has been successfully identified, as it can be seen in figure 5.3(b). However, the second loop of the original model does not yet appear, and instead the second state still has a self-loop on $cid!$, since some of its subsuming vertices in the trace tree are still incompletely explored.

Finally, when learning stops after 15 executions, the model in figure 5.3(c) is obtained, which is identical with the precise model. However, we do not consider this model a precise one, due to the fact that its construction process does not guarantee its precision. As the black box is better and better explored, it is natural for the resulting models to converge, sometimes quite early, to a model identical to the precise one. But no guarantees of precision actually exist for these models, since the trace trees out of which they were obtained had unobserved controllable transitions artificially cut out, and unobserved, but still feasible, uncontrollable transitions artificially added.

Sometimes, the complete-like approximate model can be obtained by the learning algorithm using less executions that would be needed to learn a complete model. However, in other cases, the preprocessing of the trace tree that takes place before approximate model building can actually delay convergence. This happens because the pruned controllable transitions and added uncontrollable feasible transitions, which only appear in the incomplete learning process, can interfere with the equivalence partitioning.

SSO Identity Provider – cycle based optimization

In the case of the SSO Identity Provider, the cycle-based optimization does not bring any actual improvement to the learning process. This is understandable because of the trivialty of the model, which is very easily learned by `BASYL`.

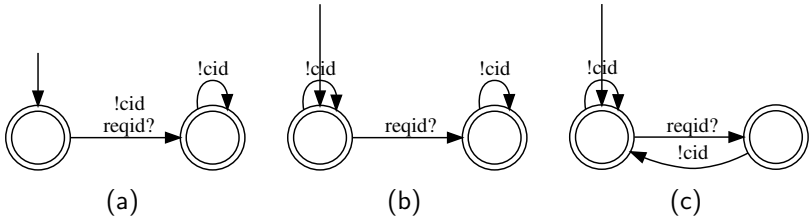


Figure 5.4: Learned models for SSO identity provider in: (a) 5 executions, (b) 10 executions, (c) 20 executions.

The models obtained can be seen in figure 5.4. They show a high similarity with the models learned by the original breadth-first exploration process, so the learning is little influenced in this case. And actually, the influence it has is for the worse, since the result converges to the precise model only after 20 executions. This happens because cycle-based learning prioritizes the cycles in the order in which they appear in the cycle specification file. Only after the traces corresponding to the highest priority cycle are completely explored in the trace tree, only then does the main focus switch to the next cycle in terms of priority.

The cycles specification for the SSO Identity Provider has the cycle $< reqid? >:< cid! >$ as the highest priority one, followed by $:< reqid? >< cid! >$. This leads to a slightly delayed discovery of the latter loop.

SSO Service Provider

When learning in isolation the model of the SSO Service Provider the values of the relevant parameters were $m = 4$ for the size of longest acyclic trace, and $\theta = 12$, the value the fairness bound.

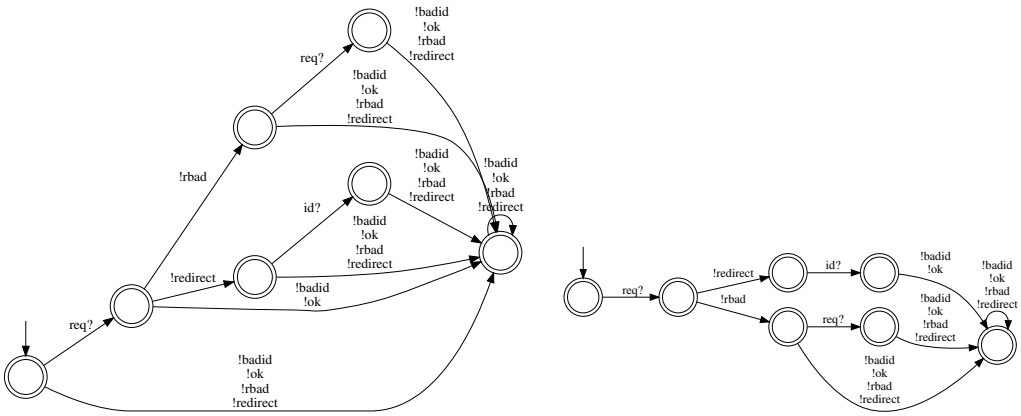
In order to learn a complete model of the SSO Service Provider the average number of executions needed was 66.7, obtained from 10 different, precise model learning experiments.

In figures 5.5 and 5.6, the learning process for the SSO Service Provider component is detailed for several considered bounds on the number of allowed executions. We can see the model learned after 5 executions in figure 5.5(a), where

the poor exploration of the black box results in many transitions to the unknown-future state (the state that has self-loops on all uncontrollable events). This is normal: as the fairness bound θ is 12, not even the initial state gets completely explored in 5 executions.

After 25 executions – figure 5.5(b) – it can be observed that several less deep states have now reached the threshold of $\theta = 12$ visits, having their infeasible uncontrollable transitions now pruned out. The three deeper states, however, still have their uncontrollable transitions towards the unknown-future state.

In figure 5.6(a) we can see an approximate model of the SSO Service Provider, learned in 50 executions. The states are better individualized, more in-depth controllable transitions got explored. However, there are still three uncontrollable transitions towards the unknown-future state, and no cycle in the original model has actually been identified.



(a) (b)

Figure 5.5: Learned models for SSO service provider in: (a) 5 executions, (b) 25 executions.

Further on, in figure 5.6(b) we have the approximate model learned in 75 executions. One of the cycles in the original model appears, however it is only partially correct (event $rbad!$ after $req?$ should have taken back into the initial state). An interesting thing is that the number of uncontrollable transitions to the unknown-future state is now slightly larger, even though the behaviour was better explored, because the equivalence partitions corresponding to the FSM states may contain vertices on a greater depth, vertices incompletely explored, which still have a large number of feasible uncontrollable transitions. After 100

executions, in figure 5.6(b) we see a better defined model, where some of the unobserved uncontrollable transitions were found infeasible and pruned out.

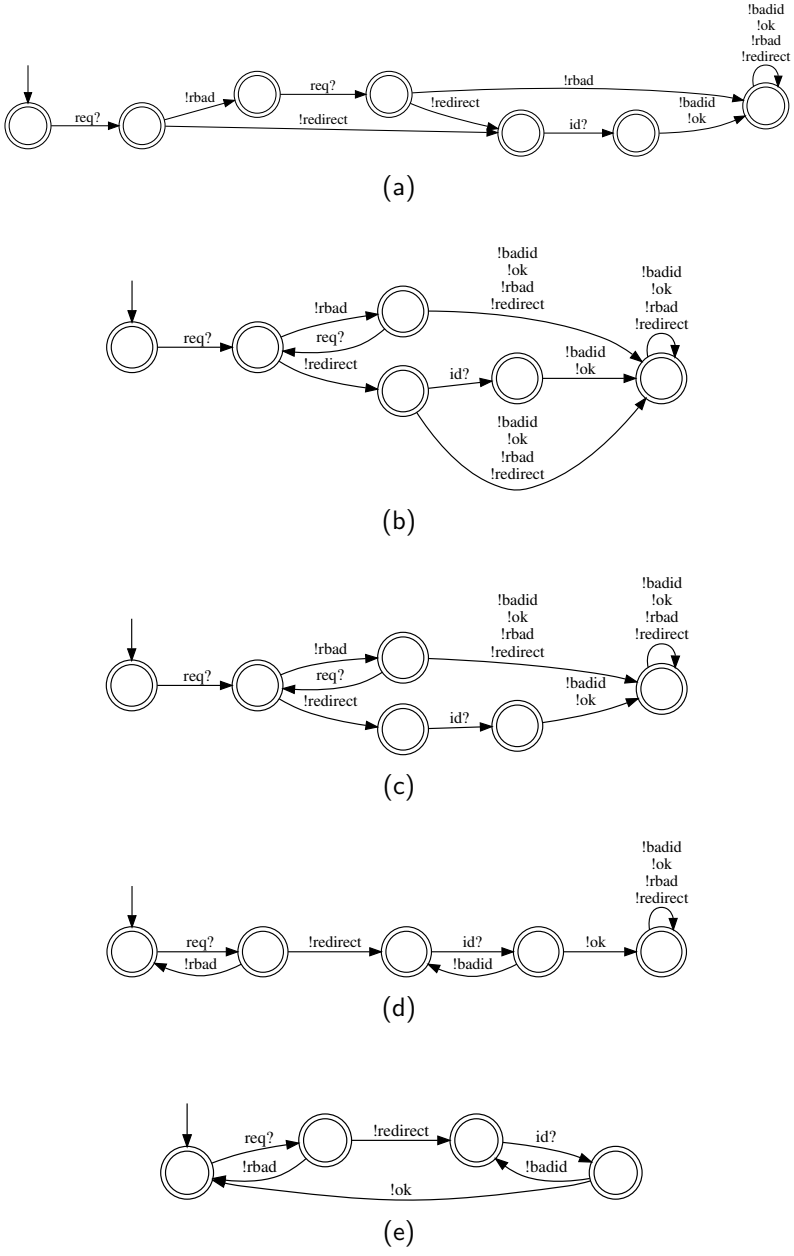


Figure 5.6: Learned models for SSO service provider in: (a) 50 executions, (b) 75 executions, (c) 100 executions, (d) 150 executions, (e) 200 executions

In figure 5.6(d), after 150 executions, the model obtained for the SSO Service Provider has two correctly identified cycles of the original model, and only one transitions to the unknown-future state, on event $ok!$, which should actually have taken the execution back to the initial state. As it is the deepest cycle in the model, this cycle was not yet identified, due to an insufficient exploration of the black box behaviour. This latter loop is finally identified after 200 runs, in figure 5.6(e), and the model converges to the original component model.

One should note that the complete model gets learned in 66.7, while the approximate model learning algorithm needs between 150 and 200 executions to converge to a similar model. This happens because, when learning a model completely we rely on the consistency-check phase to identify and test differentiating traces between compatible vertices. If such differentiating traces are not found, or are invalidated by testing, the vertices remain in the same equivalence partition, resulting in the same model state. However, when learning approximate models, because of the need to ensure a safe approximation by pruning unobserved controllable transitions and adding unobserved, but feasible uncontrollable transitions, these artificially added and/or pruned transitions at depths greater than $m + 1$ keep otherwise compatible vertices from ending up in the same partition. This is why, in some cases, obtaining a precise-like approximate model can be significantly slower than inferring a precise model.

SSO Service Provider – cycle-based optimization

In figures 5.7 and 5.8 we can see how the learning process for the SSO Service Provider takes place when cycle-based optimization is used. The early results of cycle-oriented learning, after a number of 5 and, respectively, 25 executions can be observed as identical to the ones obtained at similar stages by the unoptimized learning process (figure 5.7).

However, later on, the cycle-oriented learning seems to show visible improvement over the original learning process. Thus, the models in figure 5.8 show a more rapid convergence to the precise model, which is understandable since the cycles defined in the cycle specification file are, as we can see below, definitory for the SSO Service Provider Behaviour – with the exception of : $[1] < req? >< redirect! >< id? >< rbad! >$ which does not appear in the precise model. This means that their early identification by a cycle-focused exploration is also likely to steer an early convergence.

```
<req?><redirect!>:<id?><badid!>
```

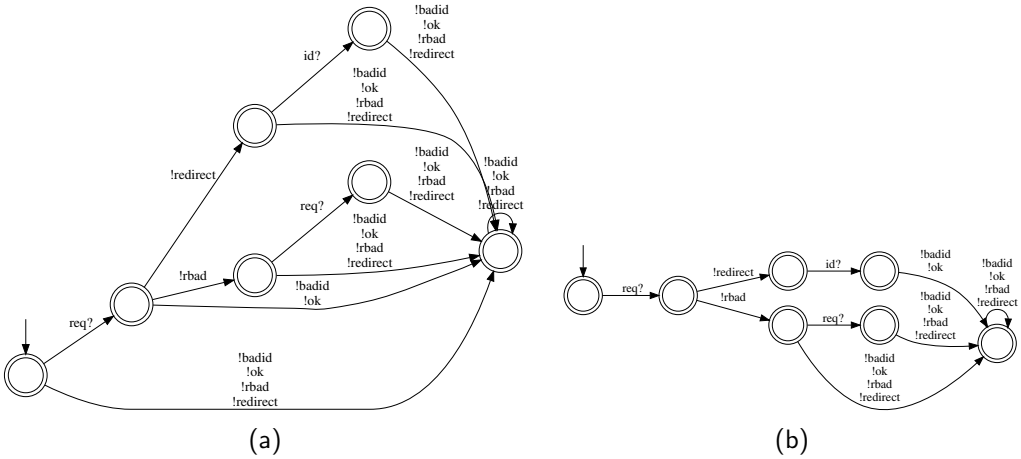


Figure 5.7: Learned models for SSO service provider in: (a) 5 executions, (b) 25 executions.

```
:[1]<req?><redirect!><id?><ok!>

:[1]<req?><redirect!><id?><rbad!>

:[1]<req?><rbad!>
```

It takes less than 50 executions (as opposed to between 50 and 75 in the original exploration case) for the learned model to reach the stage at figure 5.8(a), where one of the original model cycles is, although imperfectly, identified.

After 75 runs, the model learned looks the same as after 50 runs, however, after 100 executions (see figure 5.8(b)) another of the cycles in the original models is approximately identified, due to a focused exploration of $< req? >< redirect! >:< id? >< badid! >$, the highest priority cycle in the cycle specification file. The cycle described as : $[1] < req? >< redirect! >< id? >< ok! >$, present in the precise model is still not identified, mainly because the state targeted by the transition on event $ok!$ is still insufficiently explored, and hasn't yet been found as identical to the initial state. Also, an interesting thing to observe is that the resulting model after 100 runs has more unobserved uncontrollable transitions still considered feasible than it is the case for the normal learning process. This is a side-effect of cycle-oriented exploration, which is less uniform than normal exploration and may delay completing the exploration of some greater-depth vertices.

Finally, the learning converges to the precise-like model after at most 150 runs (see figure 5.8(c)), as opposed to requiring between 150 and 200 runs as it was the case for unoptimized exploration. So, in this case, the cycle-based exploration leads to a faster convergence than normal exploration, but does so by a relatively small margin.

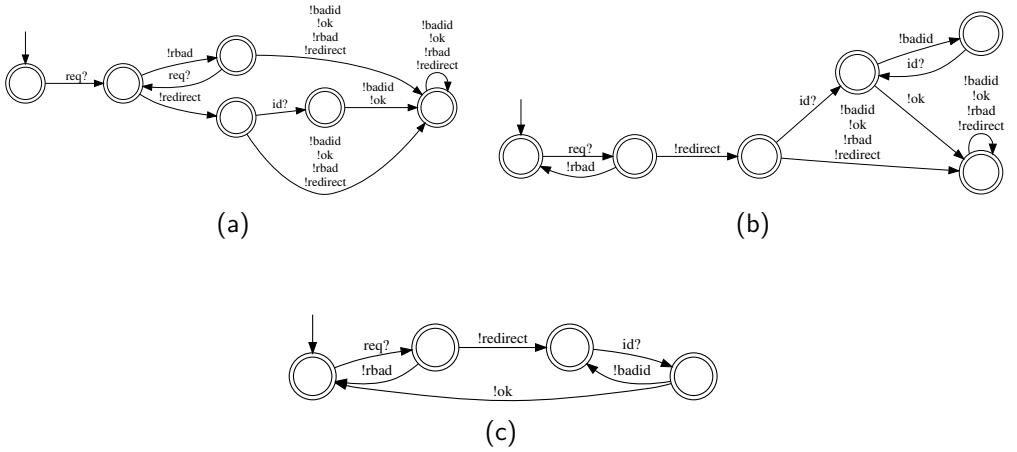


Figure 5.8: Learned models for SSO service provider in: (a) 50 executions, (b) 100 executions, (c) 150 executions.

SSO Client

The model of the SSO Client component is the largest from this case study, having the size of longest acyclic trace $m = 6$, however its alphabet has less uncontrollable events than the alphabet of the SSO Service Provider and thus the value of the fairness bound is relatively smaller here, being only $\theta = 8$.

In order to learn a complete model of the SSO Client the average number of executions needed was 62.5, obtained from 10 independently performed experiments. This value is significantly low for a model of this size, and smaller than the one required to learn the SSO Service Provider. This happens because the SSO Client not only has a low fairness bound and an overall small alphabet, but it is also deterministic in its behaviour: no more than one uncontrollable event does fire from any of its state.

The results of the learning process can be seen in figures 5.9 and 5.10. After 5 executions, the behaviour of the SSO Client is little explored, and the number of

$\theta = 8$ visits hasn't been met for any of the model states, not even for the initial one, as we can observe from all the uncontrollable transitions to the unknown-future state (see figure 5.9(a)). After 10 executions, the fairness bound has been met only for the initial state, as it can be seen from figure 5.9(b). In figure 5.9(c) we can see that, due to the fact of SSO Client has a deterministic and relatively simple behaviour, a number of 25 executions has sufficed for vertices at depths from 0 to 3 to be completely explored.

Further on, in figure 5.10(a), after 50 executions, the model obtained shows a relatively deep exploration of the component behaviour, as no incompletely explored states appear, however the exploration depth is yet insufficient for cycle identification. After 75 executions, as it can be seen in figure 5.10(b), two of the cycles in the precise model of the SSO Client have been identified.

As the number of runs continues to increase, vertices at greater depths are found to be equivalent with more shallow ones. Such vertices then become part of the same equivalence partition, but their outgoing transitions might lead to vertices less explored, which end up in different equivalence partitions. As each equivalence partition emerges as a state in model, this situation might end up in nondeterminism, as it is the case for the automaton in figure 5.10(c), the result of 150 learning executions, where we have two transitions on the same event $rbad?$ from the second state of the FSM. After 250 runs, the remaining cycle of the original model has finally been identified (see figure 5.10(d)), but a case of nondeterminism similar to the one above can be observed, as we have two transitions on event $req!$ from the initial state. Finally, after 500 executions, a precise-like model is obtained (see figure 5.10(e)).

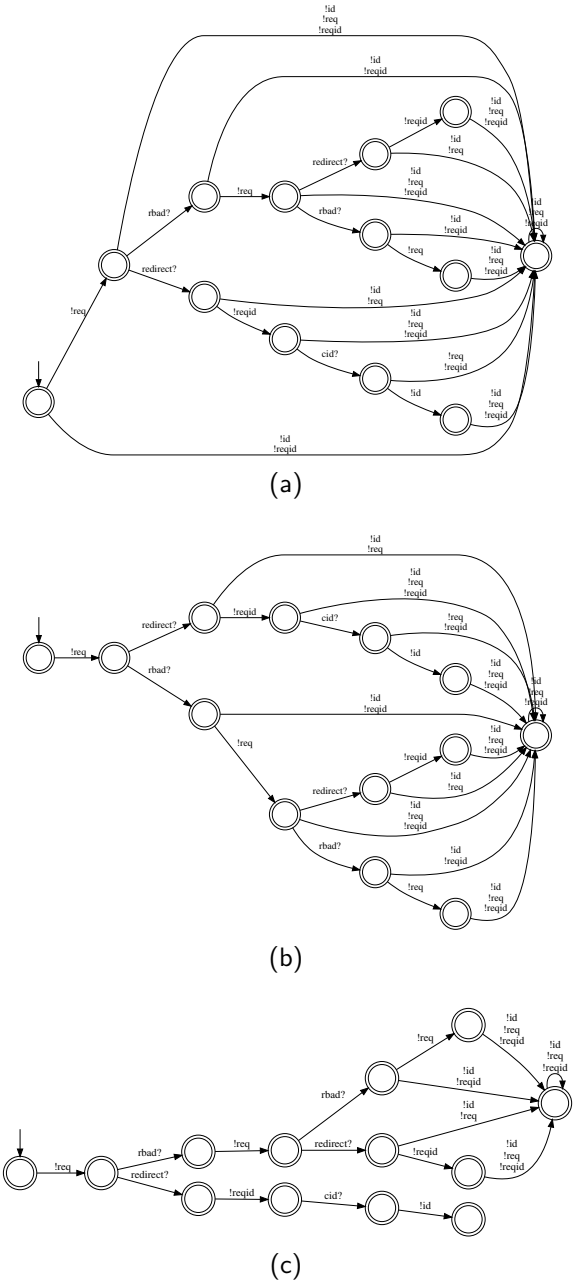
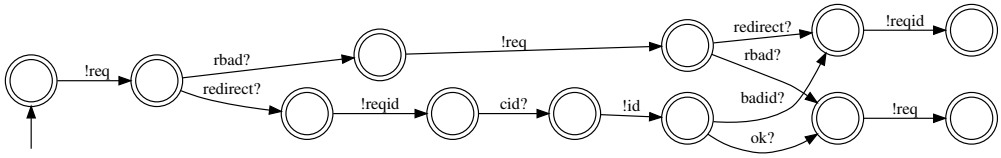
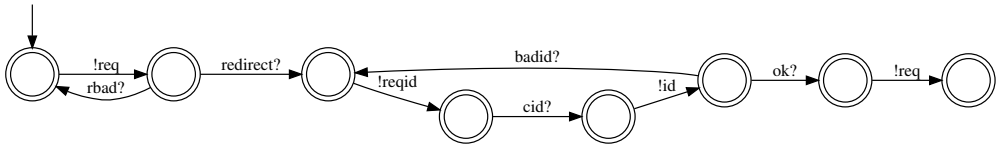


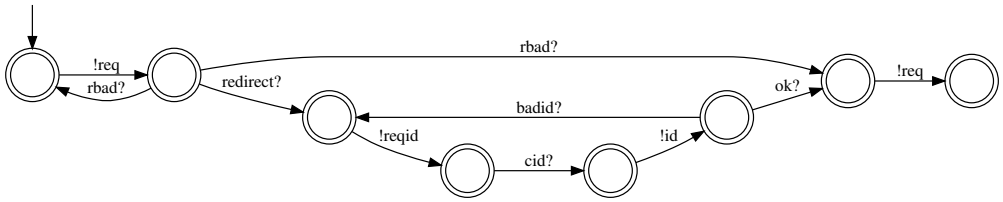
Figure 5.9: Learned models for SSO client in: (a) 5 executions, (b) 10 executions, (c) 25 executions.



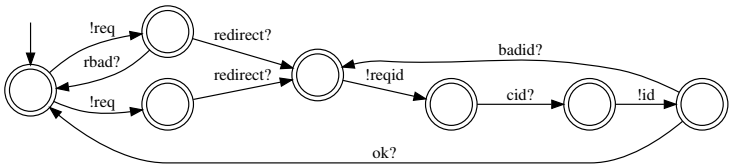
(a)



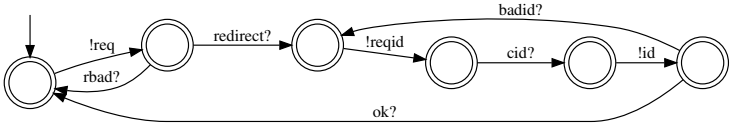
(b)



(c)



(d)



(e)

Figure 5.10: Learned models for SSO client in: (a) 50 executions, (b) 75 executions, (c) 150 executions, (d) 250 executions, (e) 500 executions.

SSO Client – cycle-based optimization

The optimization of the learning process with an emphasis on cycle-identification seems to prove beneficial in the case of learning the SSO Client component in isolation. The results of the learning process can be seen in figures 5.11 and 5.12, while the specified cycles are given below (all but the third cycle actually appear in the precise model).

```
<req!><redirect?>:<reqid!><cid?><id!><badid?>

:[1]<req!><redirect?><reqid!><cid?><id!><ok?>

:[1]<req!><redirect?><reqid!><cid?><id!><rbad?>

:[1]<req!><rbad?>
```

In this specific case, the early results of cycle-oriented learning already exhibit some differences from unoptimized learning. Thus, the models at figures 5.11(a) and 5.11(b), obtained after a number of 5 and, respectively, 10 executions have a significantly smaller number of states – 7, than their correspondents at figures 5.9(a) and 5.9(b), resulted from the original learning algorithm, and which both have 13 states. This happens because as the exploration was focused on the specified cycles, their corresponding traces were observed more times, leading to more completely explored vertices than by uniform exploration. This effect is also visible in figure 5.11(c) which, although quite similar to the one in figure 5.9(c), has less transitions towards the unknown-future state.

However, as the cycle-based learning process continues, we can observe that the models obtained after 50 and 75 executions, shown in figures 5.12(a), respectively 5.12(b), are identical to the ones learned by uniform exploration from figures 5.10(a), respectively 5.10(b). At this stage, in both cases, the vertices at depths less than m are already completely explored and cycles are identified when vertices at depths $m + 1$ also get completely explored and found as equivalent to their more shallow peers.

After 150 executions, the model in figure 5.12(c) is obtained which, with the unoptimized learning algorithm was obtained in more than 150, less than 250 runs. The precise-like model of the SSO Client is learned after at most 250 executions (see figure 5.12(d)), while being converged to in more than 250 and less than 500 executions in the unoptimized setting.

Therefore, the cycle-based learning algorithm tends to be more efficient in identifying the specified cycles in the model, which leads to a faster convergence to the precise model if all, or most of the model cycles appear in the cycle specification file. However, in this case, the margin of efficiency that the cycle-based learning has over unoptimized learning is not large (as models obtained after 50 and 75 are identical, etc.), which is understandable as SSO Client is an easy to learn component. We did not try to identify this margin exactly, as it would have implied a significant number of additional experiments. Also, the result obtained would have been very specific to the experiment at hand, and a general such margin would have to consider many parameters such as the size of the model, of its alphabet, its fairness bound, the degree of nondeterminism in the model, the number of specified cycles, their depth, how do they cover the model transitions, the order in which they are specified, etc.

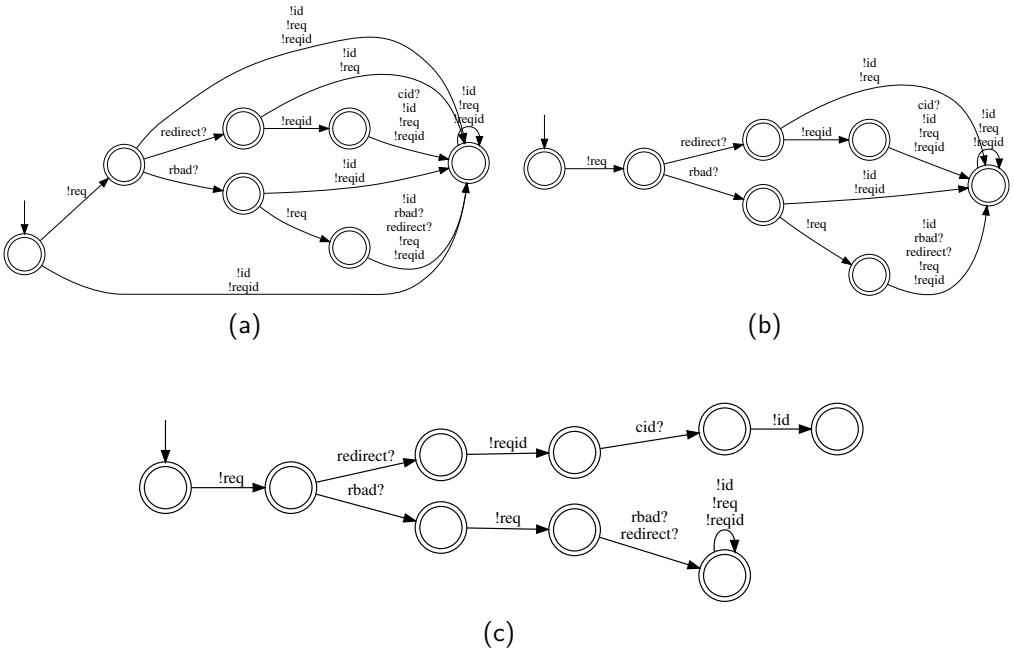
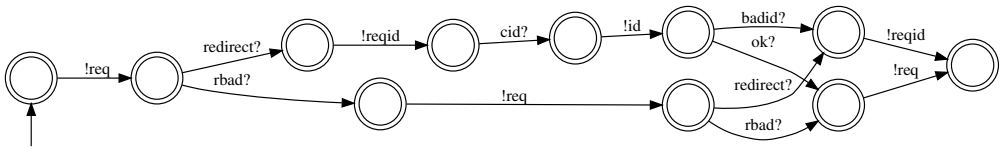
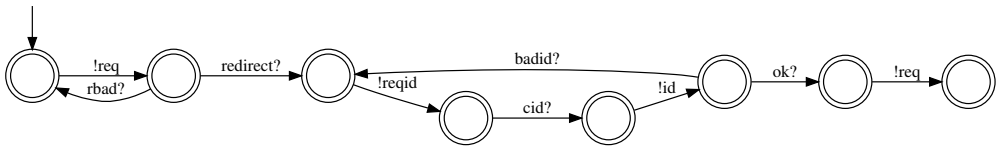


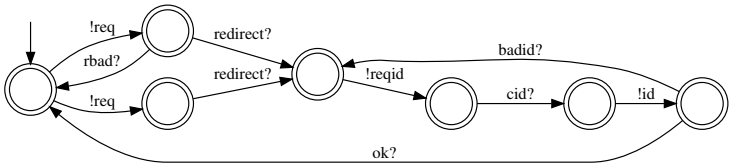
Figure 5.11: Learned models for SSO client in: (a) 5 executions, (b) 10 executions, (c) 25 executions.



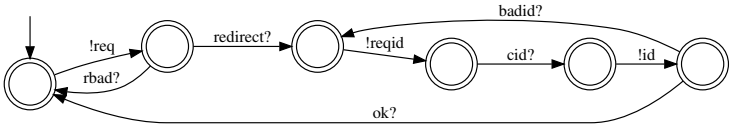
(a)



(b)



(c)



(d)

Figure 5.12: Learned models for SSO client in: (a) 50 executions, (b) 75 executions, (c) 150 executions, (d) 250 executions.

5.2.2 Centralized Exploration

The centralized exploration is the closest emulation of how the components will work together in the system, with the difference that the proactive adaptor controls all interactions in the learning executions, steering each time the ongoing run towards one or another interesting execution scenario. To avoid exploring behaviour actually irrelevant to the system we want to build, only the executions that conform to the specified safety property (see figure 5.2) are explored.

The learning process that relies on centralized exploration has several specifics that distinguish it both from learning components in isolation and from learning based on distributed exploration.

First, there is only one global execution trace, from which the individual component event traces to be used in learning are obtained by projection on each specific component's event set. An execution is normally stopped when it reaches a maximum number of steps, but in the centralized setting an execution is only stopped when each of its projection traces have reached or surpassed the maximum number of steps. This leads to significantly longer execution traces than in the distributed setting. Also, the global traces end up having variable sizes. This slows down the inference process.

Second, as the interactions are coordinated in a centralized way, it is difficult to generate traces that would explore each component sufficiently well. Even though the model of each component is built separately, from separate trace trees, exploring the behaviour of these components together amounts to exploring the global behaviour of the system, which is hard.

Third, as far as cycle-based exploration is concerned, it is important that the cycles specified for each component that are projections of the same global property cycle to appear in the same order, and thus have similar priorities, since there is only one global execution being coordinated.

For the learning experiments performed on the Single Sign On case study we have a global size for the longest acyclic path, which is $m = 6$, while the fairness bounds are set independently for each component, thus $\theta = 2$ for the SSO Identity Provider, $\theta = 12$ for the SSO Service Provider and $\theta = 8$ for the SSO Client. After each learning experiment, a controller has been computed for the inferred models and the desired safety property. We have then verified whether the obtained controller did work on the real system, i.e. on the plant determined by the precise models of participating components. For controller synthesis and verification we have used the Supremica tool [59].

Normal Exploration

In the following we will present the results of learning the models of the SSO entities using the centralized exploration algorithm. Table 5.1 summarizes the statistical data characterizing these results, i.e. the inferred models and the generated system controllers. The data presented represents the number of

states and number of transitions of all automata.

It is important to note that, while the generated controllers are always already minimized, this is not the case for the learned models. The models built by BASYL are not always minimal, due to the safe approximation strategy, which forces the pruning, etc. of the trace tree, and thus keeps equivalent, but incompletely explored vertices from being in the same equivalence partition. Therefore, we also minimize the learned models before controller generation and sometimes it is only after this minimization that a model becomes precise-like. This is why in table 5.1, as in all similar tables from this work, we present first the statistical data of the unminimized model, then, on the following row, the data for the minimal model.

Some of the models obtained during the centralized learning process, after at most 1000 executions, can be seen in figures 5.13 – the SSO Identity Provider, 5.14 – the SSO Service Provider and 5.15 – the SSO Client.

From table 5.1 we can see that after 100 runs, the models obtained for the SSO Service Provider, the SSO Identity Provider and the SSO Client are rather large, even after minimization. Looking at the figures at 5.13(a), 5.14(a) and 5.15(a) we can also see how these models look. The behaviour of the SSO Identity Provider, as seen in figure 5.13(a), is quite well explored. It could look like one of its important cycles has been identified, but it is actually the unknown-future state that has a self-loop on the only uncontrollable event in the alphabet. The model is far from converging to the precise-like automaton. This happens mainly because the value of m has been globally set to 6, while its actual value is 2. In the case of the SSO Service Provider, whose learned model can be seen in figure 5.14(a), the behaviour of the component has been completely explored for all visible states, except from the deepest two, however, no cycle has been identified in the model. Here, the assumed value of $m = 6$ is also larger than its actual one, of 4, which may also delay cycle identification. The SSO Client is the best explored of all three, and two of its three cycles are already identified (see figure 5.15(a)). But, due to the other two learned models, the obtained controller, found as having 796 states and 2882 transitions, is pretty restrictive on the real system, not allowing for repeating interaction scenarios.

After 250 executions, all models inferred seem significantly better explored and one of them has already converged to the precise-like model, as it can be seen from both 5.1 and figures 5.13(b), 5.14(b) and 5.14(b). The two cycles have been identified for the SSO Identity Provider, although only one is precisely identified, and the SSO Service Provider also has one identified loop – both

these components still have their deepest states incompletely explored. The SSO Client has its third cycle identified, has converged to the precise-like model and from table 5.1 it results that the learning will deliver the precise-like model in all further experiments.. However, the obtained system controller, with 390 states and 1706 transitions only allows an unlimited number of repetitions to the scenario that describes the rejection of a badly formulated request.

After 500 executions, as it can be seen in 5.13(c), the SSO Identity Provider has already converged to a precise-like model with 2 states and 3 transitions (when minimized) and from table 5.1 we can see that it will also converge to the precise-like model in all further experiments. The SSO Service Provider in figure 5.14(c) is quite close to converging, all its cycles have been identified and only two of its states still have transitions to the unknown-future state. The SSO Client component has previously converged. Thus, the computed controller (having 245 states, 1137 transitions in table 5.1) allows us an actually complete use of all the scenarios in the specification, which are allowed to repeat in any order, for an unlimited number of times.

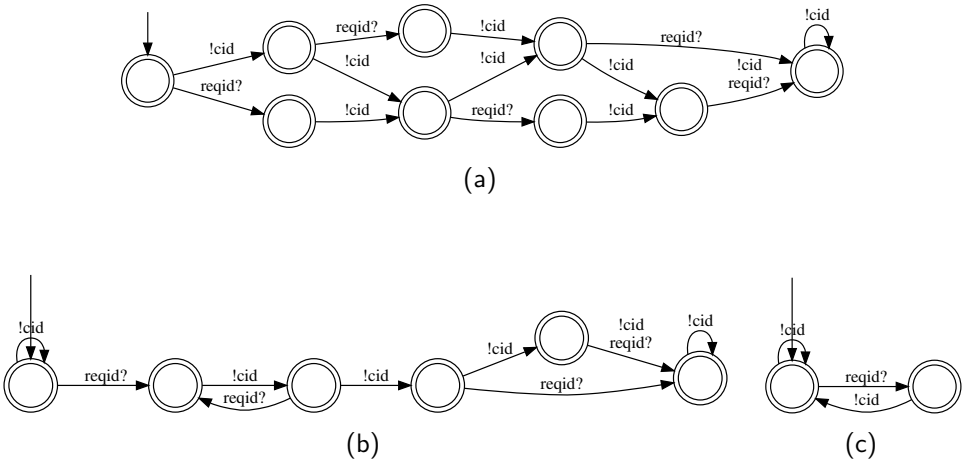
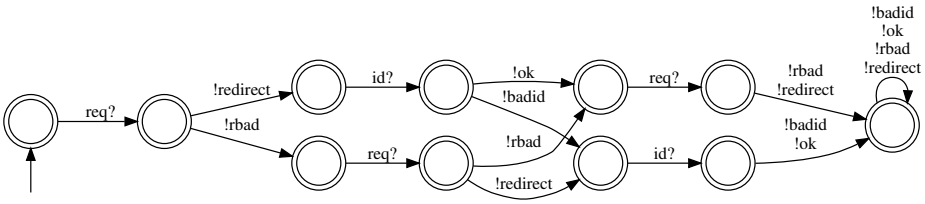
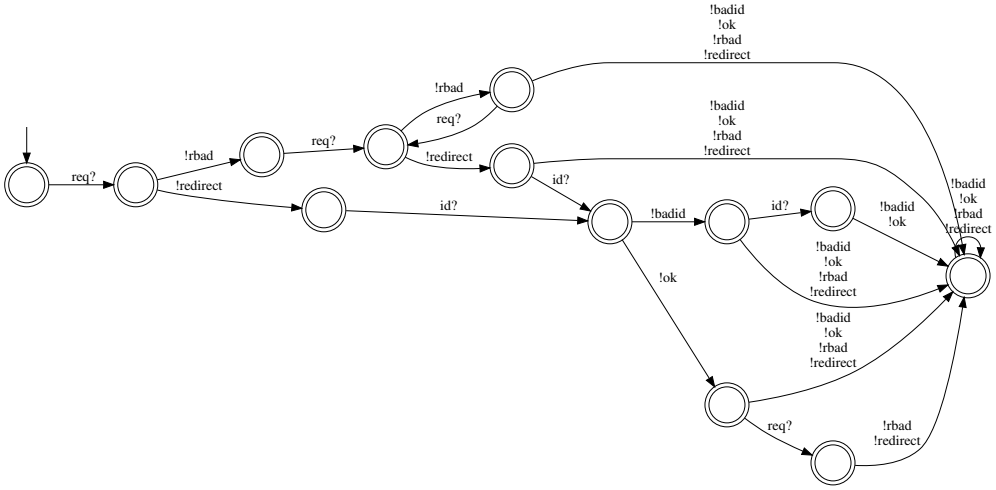


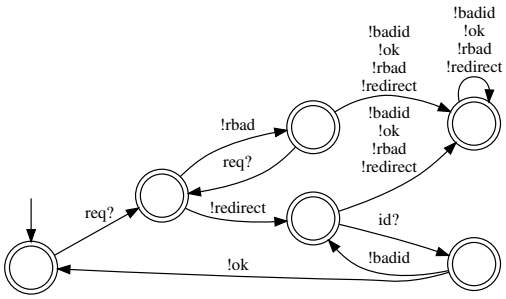
Figure 5.13: Learned models for SSO identity provider in: (a) 100 executions, (b) 250 executions, (c) 500 executions.



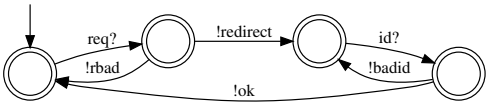
(a)



(b)

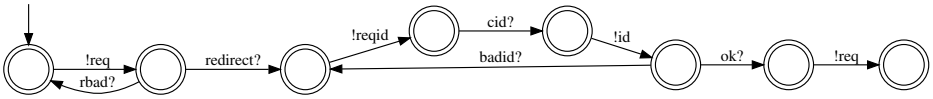


(c)

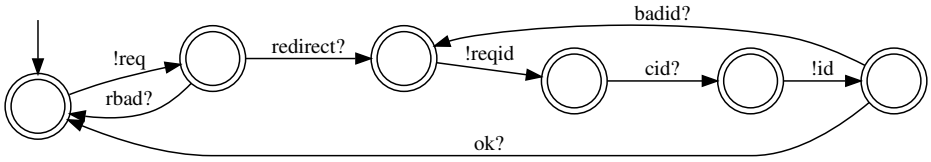


(d)

Figure 5.14: Learned models for SSO service provider in: (a) 100 executions, (b) 250 executions, (c) 500 executions, (d) 1000 executions.



(a)



(b)

Figure 5.15: Learned models for SSO client in: (a) 100 executions, (b) 250 executions.

Further on, when experimenting with a bound of 1000 executions, the SSO Service Provider component also converges to its precise-like model, having 4 states and 6 transitions, as it can be seen from figure 5.14(d). The SSO Identity Provider model and the SSO Client component, as earlier mentioned, also converge. Thus, all models inferred are precise-like and the 276 states and 1180 transition controller obtained for these models is identical to the one obtained for the real models.

The average trace length encountered during the normal exploration process was 46.6. This is significantly more than $3 \times 2m = 36$ – the ideal length of an execution trace for 3 entities with $m = 6$, and is a relevant drawback: runtime executions are considered expensive because the communication with the component is expensive, so the need for longer execution traces naturally reflects in higher learning costs.

Table 5.1: Model inference results

nr.	Cl. s.	Cl. t.	SP s.	SP t.	IDP s.	IDP t.	ctrl s.	ctrl t.
100	10 8	13 9	16 11	25 19	34 9	45 14	796	2882
250	7 6	10 8	14 13	25 37	29 6	43 10	390	1706
500	6 6	8 8	10 6	22 19	23 2	38 3	245	1137
1×10^3	6 6	8 8	4 4	6 6	11 2	20 3	276	1180
1.5×10^3	6 6	8 8	4 4	6 6	8 2	13 3	276	1180
2×10^3	6 6	8 8	4 4	6 6	6 2	9 3	276	1180
2.5×10^3	6 6	8 8	4 4	6 6	5 2	7 3	276	1180
5×10^3	6 6	8 8	4 4	6 6	2 2	3 3	276	1180
1×10^4	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	..	..
1.5×10^4	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	..	..

Cycle-Based Exploration

The cycle-based optimization of the algorithm for model learning by centralized optimizations does provide some improvement over the results in the previous section, as the models for the SSO Identity Provider and the SSO Service Provider do improve and converge earlier to precise-like automata. The model learning process for the SSO Client component results in results similar to the ones obtained by unoptimized centralized exploration. The statistical results of the model learning process are summarized in the table 5.2, while the models obtained for the first 4 experiments registered in the table are present in figure 5.16 for the SSO Identity Provider, figure 5.17 for the SSO Service Provider, and figure 5.18 for the SSO Client.

As mentioned, the model for the SSO Identity Provider does seem to converge slightly faster than in the unoptimized setting, and we can see from figure 5.16(b) that it has already converged to the precise-like model in no more than 250 runs. The SSO Service Provider, although it converges to the precise-like model in no more than 1000 runs, does obtain better intermediary results: a better explored, cleaner model after 100 runs (see figure 5.17(a)), two identified cycles instead of one after 250 runs (figure 5.17(b)), and more precisely identified cycles after 500 runs (figure 5.17(c)). Thus, for these two entities, the cycle-based optimization shows noticeable improvements over the unoptimized algorithm.

The average trace length encountered during the cycle-based exploration process was 47.46, only a little more than the average trace length of 46.6 observed for normal centralized exploration. Since both average values have been obtained by randomly choosing 30 sample trace lengths (from the log files corresponding to the 100 runs experiment), we consider a margin of 0.86, representing no more than 0.18% of their mean value, as irrelevant.

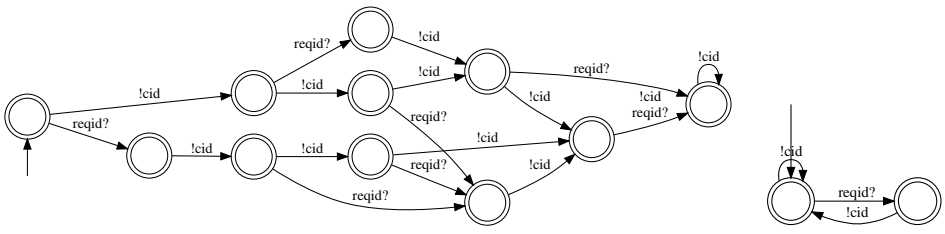
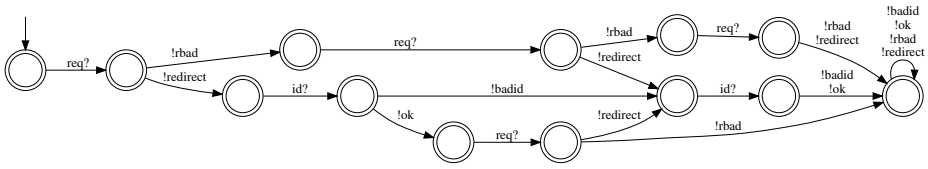
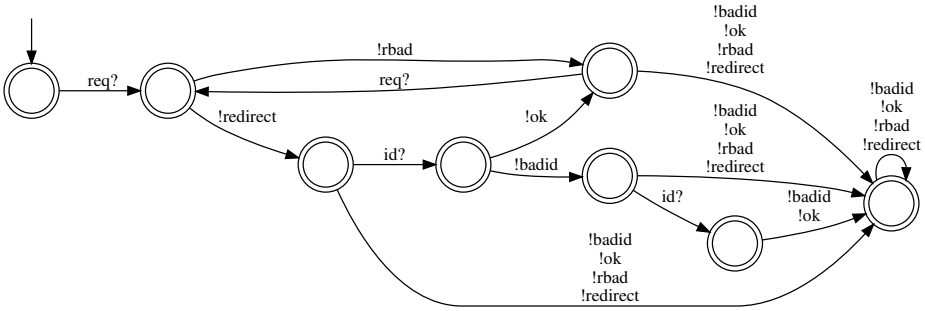


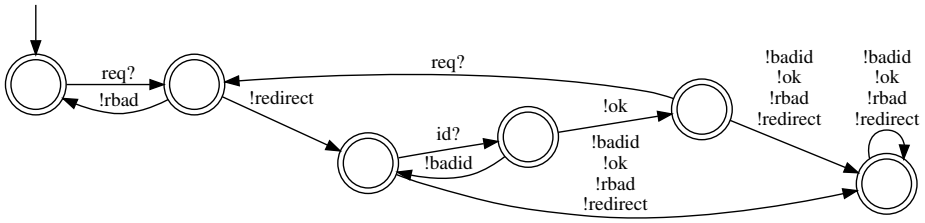
Figure 5.16: Learned models for SSO identity provider in: (a) 100 executions, (b) 250 executions.



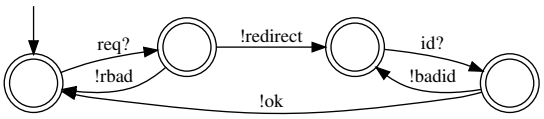
(a)



(b)



(c)



(d)

Figure 5.17: Learned models for SSO service provider in: (a) 100 executions, (b) 250 executions, (c) 500 executions, (d) 1000 executions.

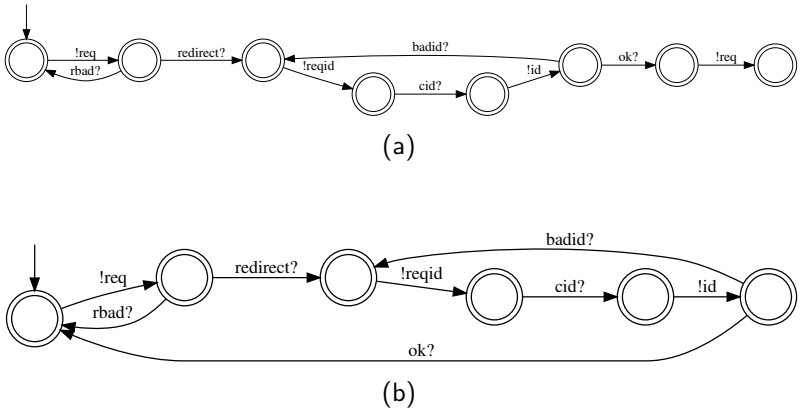


Figure 5.18: Learned models for SSO client in: (a) 100 executions, (b) 250 executions.

Table 5.2: Model inference results

nr.	Cl. s.	Cl. t.	SP s.	SP t.	IDP s.	IDP t.	ctrl s.	ctrl t.
100	10 8	13 9	16 13	25 22	34 11	46 18	1252	4789
250	6 6	8 8	13 8	25 26	29 2	42 3	255	1163
500	6 6	8 8	8 6	16 19	21 2	35 3	245	1197
1×10^3	6 6	8 8	4 4	6 6	12 2	20 3	276	1180
1.5×10^3	6 6	8 8	4 4	6 6	9 2	15 3	276	1180
2×10^3	6 6	8 8	4 4	6 6	5 2	6 3	276	1180
2.5×10^3	6 6	8 8	4 4	6 6	6 2	11 3	276	1180
5×10^3	6 6	8 8	4 4	6 6	2 2	3 3	276	1180
1×10^4	6 6	8 8	4 4	6 6	2 2	3 3	276	1180
1.5×10^4	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	..	..

5.2.3 Distributed Exploration

The distributed exploration setting allows for parallelism in the model inference process. Each component has a local proactive adaptor, which intercepts its sent messages and coordinates the execution by forwarding messages to the component, in order to enable its controllable transitions. Messages sent by a component and intercepted by the proactive adaptor are broadcast to the rest of the components in the system. The proactive adaptor also intercepts the messages broadcast by the other, remote components.

To only explore behaviour relevant to the system we want to build, the executions that conform to the specified safety property (see figure 5.2) are exclusively considered. Unlike the centralized case, which had all real components interact together, in the distributed case the environment of the local component is only simulated. The distributed exploration method thus explores the behaviour of the local component by steering the execution at runtime towards interesting scenarios, while the other components in the system are explored only in the model, by a bounded depth-first search phase. When some of the other components in the system are unknown, the models employed for them are most general ones, i.e. they have only one state, with self-loop transitions for all events in the alphabet.

The exploration process runs in parallel for all components in the system. The drawback of longer execution traces is thus eliminated, since each component is locally explored, so all execution traces will have a maximum size of $2m$.

The distributed exploration, however, does not assume complete independence between the parallel exploration processes, and this is what distinguishes it from the learning in isolation setting. Only messages previously broadcast by a remote component can be used by the proactive adaptor to enable controllable transitions in the local component. Thus, convergence to precise-like models is expected to be slower for distributed than for individual exploration.

Just as in the centralized exploration case, for the distributed learning experiments performed on the Single Sign On case study we established a global value for the longest acyclic path, which is $m = 6$. Again, the fairness bounds are set independently for each component: $\theta = 2$ for the SSO Identity Provider, $\theta = 12$ for the SSO Service Provider and $\theta = 8$ for the SSO Client. After each learning experiment, a controller has been computed for the inferred models and the desired safety property. Then, we have verified whether the controller obtained for the learned, approximate models did indeed work on the real system, represented

by the asynchronous product of its components' precise models. For controller synthesis and verification we have used also here the Supremica tool [59].

Normal Exploration

The results of the learning process using the unoptimized distributed exploration setting are presented in this section. Some of the models obtained by the inference process can be seen in figure 5.19 for the SSO Identity Provider component, in figure 5.20 for the SSO Service Provider and in figure 5.21 for the SSO Client component. In table 5.3 the statistical data for all the experiments performed for unoptimized local model learning are summarized. From the first glance, the results obtained are significantly similar to the ones obtained in the centralized learning setting, being also relatively close, but inferior in efficiency, to the individual exploration setting.

After 100 executions, the models at figure 5.19(a), 5.20(a) and 5.21(a) are obtained for the three components in the system. The SSO Identity Provider and the SSO Service Provider still have their deepest states incompletely explored and exhibiting transitions to the unknown-future state, while the SSO Client already has two identified cycles. This happens because the exploration of first two depends on messages sent by SSO Client, which is the component actually initiating the conversation. The controller obtained has 957 states and 3561, however it only allows for a single execution of each of the relevant scenarios.

After 250 executions, the models for component SSO Identity Provider and SSO Client have reached convergence to the precise-like automata, while the model for SSO Service Provider has 2 out of its 3 cycles identified and has 4 of its deepest states exhibiting uncontrollable transitions to the unknown-future state.– see table 5.3 and figures 5.19(b), 5.20(b), and 5.21(b). The obtained controller, as it can be seen in table 5.3, has 145 states and 660, allowing an unlimited executions of the scenarios in which a malformed request is rejected, or a bad certificate is answered with a refusal messages by the service provider, but only one execution of the scenario in which the request is successfully answered.

After 500 executions, as we can see from table 5.3 and figure 5.20(c), the models for component SSO Identity Provider and SSO Client have already converged to the precise-like automata, while the model for SSO Service Provider has all its 3 cycles identified, although one of them only approximately, and still has

two states exhibiting uncontrollable transitions to the unknown-future state. This is understandable when taking into account that the SSO Service Provider component has the largest fairness bound, 12, out of all three entities. Their corresponding controller has 255 states and 1236 transitions, and allows for all the relevant scenarios in the model to repeat unlimitedly.

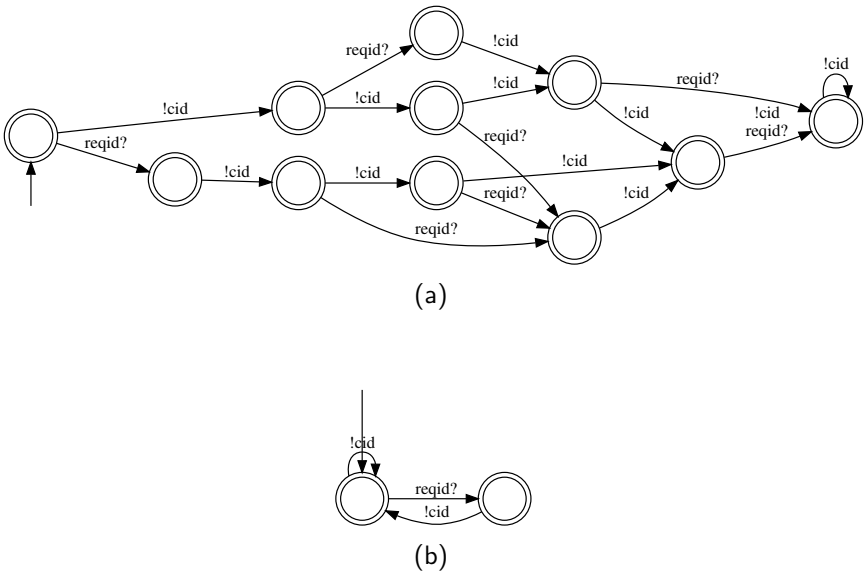
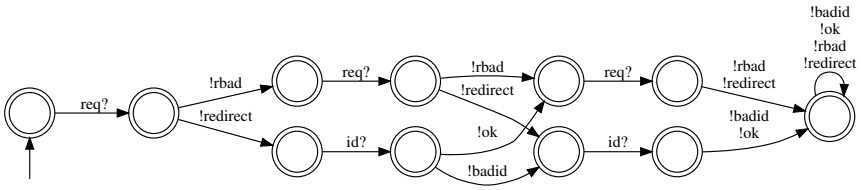
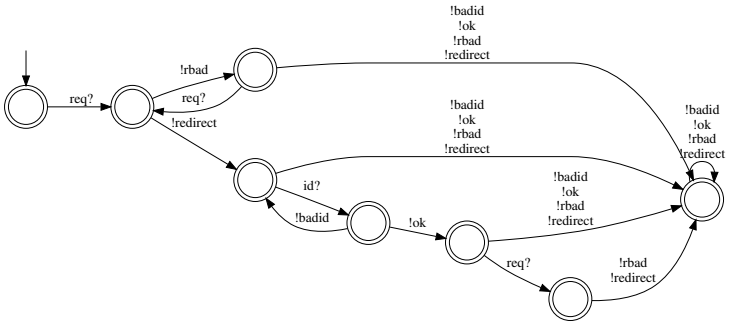


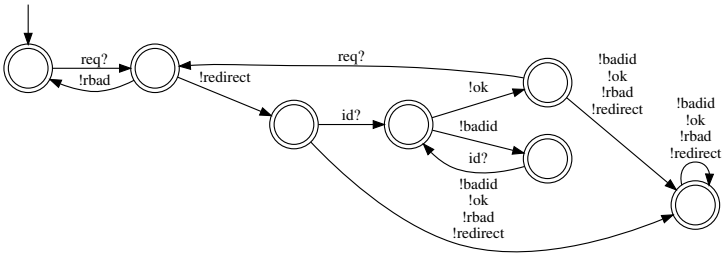
Figure 5.19: Learned models for SSO identity provider in: (a) 100 executions, (b) 250 executions.



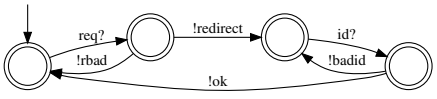
(a)



(b)

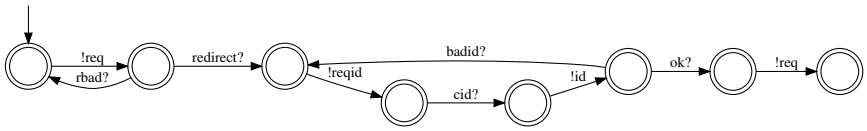


(c)

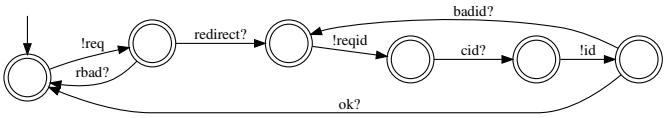


(d)

Figure 5.20: Learned models for SSO service provider in: (a) 100 executions, (b) 250 executions, (c) 500 executions, (d) 1000 executions,



(a)



(b)

Figure 5.21: Learned models for SSO client in: (a) 100 executions, (b) 250 executions.

Table 5.3: Model inference results

nr.	Cl. s.	Cl. t.	SP s.	SP t.	IDP s.	IDP t.	ctrl s.	ctrl t.
100	10 8	13 9	16 11	25 19	34 11	46 18	957	3561
250	6 6	8 8	11 8	22 26	27 2	40 3	145	660
500	6 6	8 8	8 7	16 20	21 2	34 3	255	1236
1×10^3	6 6	8 8	4 4	6 6	10 2	18 3	276	1180
1.5×10^3	6 6	8 8	4 4	6 6	7 2	14 3	276	1180
2×10^3	6 6	8 8	4 4	6 6	6 2	10 3	276	1180
2.5×10^3	6 6	8 8	4 4	6 6	5 2	8 3	276	1180
5×10^3	6 6	8 8	4 4	6 6	2 2	3 3	276	1180
1×10^4	6 6	8 8	4 4	6 6	2 2	3 3	276	1180
1.5×10^4	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	..	..

Further on, after 1000 executions, all models obtained by distributed inference have converged to precise-like models. The controller obtained in this case is the ideal controller for the real system, and it has 276 states and 1180 transitions. As it can be seen from table 5.3 and as expected, all further experiments also result in a perfect convergence to the precise-like models for all components.

Cycle-Based Exploration

The improvements resulted from the cycle-based optimization of the learning algorithm are quite negligible when comparing table 5.3 to table 5.4, where the latter one presents the summarized results for the optimized learning setting. We can see that the statistical data regarding obtained models is pretty much similar in both cases, and the models are found to converge to precise-like automata at similar milestones: after at most 250 executions for the SSO Client and the SSO Identity Provider, and after 1000 executions for the SSO Service Provider, the component with the highest fairness bound of this system.

The models resulted from the cycle-based learning in the distributed setting can be seen in figure 5.22 for the SSO Identity Provider, in figure 5.23 for the SSO Service Provider and in figure 5.24 for the SSO Client component. When not reaching convergence, the results harboured by the experiments are quite similar with the ones obtained by unoptimized learning. The only notable exception is that, in figure 5.22(a), the model of the SSO Identity Provider is smaller and already has, after at most 100 runs, one identified cycle from its initial state, while its correspondent from the set of unoptimized learning experiments doesn't have any identified cycles.

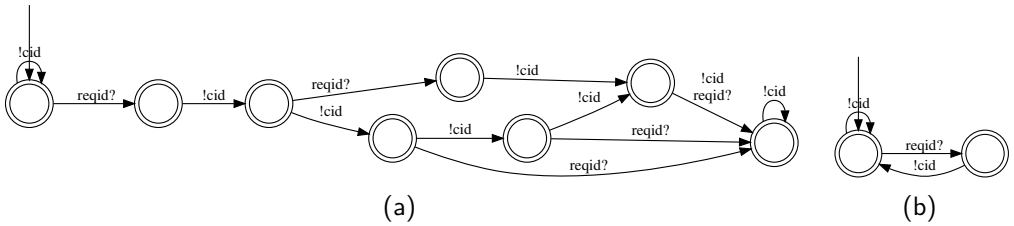
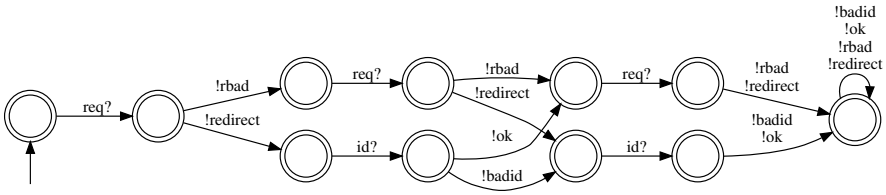


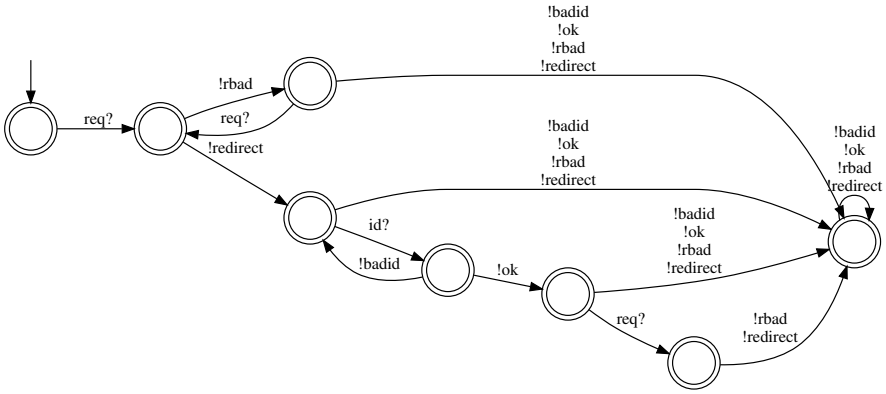
Figure 5.22: Learned models for SSO identity provider in: (a) 100 executions, (b) 250 executions.

Table 5.4: Model inference results

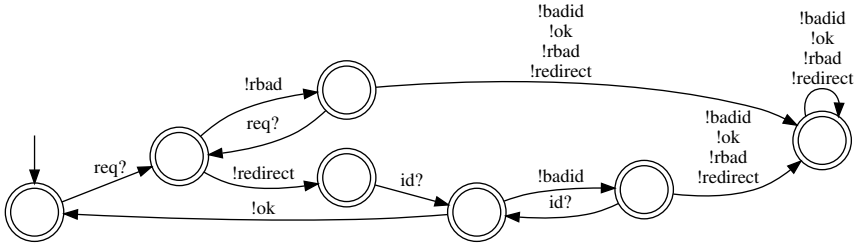
nr.	Cl. s.	Cl. t.	SP s.	SP t.	IDP s.	IDP t.	ctrl s.	ctrl t.
100	10 8	13 9	16 11	25 19	33 8	47 13	789	2920
250	7 6	10 8	13 8	25 26	28 2	41 3	145	660
500	6 6	8 8	8 7	16 20	19 2	35 3	245	1091
1×10^3	6 6	8 8	4 4	6 6	2 2	3 3	276	1180
1.5×10^3	6 6	8 8	4 4	6 6	7 2	11 3	276	1180
2×10^3	6 6	8 8	4 4	6 6	5 2	6 3	276	1180
2.5×10^3	6 6	8 8	4 4	6 6	5 2	7 3	276	1180
5×10^3	6 6	8 8	4 4	6 6	2 2	3 3	276	1180
1×10^4	6 6	8 8	4 4	6 6	2 2	3 3	276	1180
1.5×10^4	.. ..	.. ..	.. ..	.. ..	.. ..	.. ..	..	..



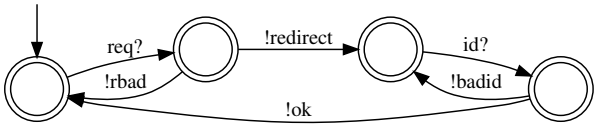
(a)



(b)

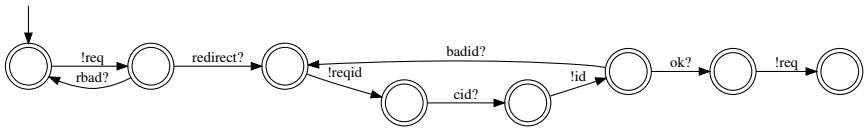


(c)

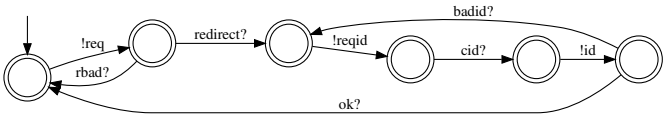


(d)

Figure 5.23: Learned models for SSO service provider in: (a) 100 executions, (b) 250 executions, (c) 500 executions, (d) 1000 executions,



(a)



(b)

Figure 5.24: Learned models for SSO client in: (a) 100 executions, (b) 250 executions.

5.3 Case Study: A Product Data Management System

The Product Data Management System is one of the case studies presented in [78]. We have built its abstract asynchronous version, as the initial case study considered in [78] was synchronous. We have chosen the case study of components ThinkTeam and CAD, and adapted it for our purposes, by using only one CAD instance instead of two, and by rewriting the coordination policy accordingly (see figure 5.25).

Think Team is a Product Data Management (PDM) platform, that provides its users with features to manage documents, versions, data attributes, and relationships among documents, which is to be integrated with a CAD application. Together, the two components will form a Product Data Management cooperative system, that will allow for a better, more flexible management of all product documentation.

The alphabet of the ThinkTeam and CAD components contains send and receive events of the following messages:

- start: starts the integrated ThinkTeam component
- stop: stops the integrated ThinkTeam component

- afterinit: received by ThinkTeam when its partner is completely initialized
- ttready: sent by ThinkTeam when it is completely initialized
- checkin: locks a specified file into the centralized repository for writing operations
- checkout: releases a previous lock on a specified file into the centralized repository for writing operations
- get: gets a local copy of a file
- import: stores a local file into a centralized repository
- openfile: opens a specified file
- closefile: closes a specified file
- save: saves a specified file
- remove: removes a specified file from the centralized repository
- setvalue: sets or modifies the value of a certain attribute of a file in the centralized repository
- setvalues: sets or modifies the values of all attributes for a file in the centralized repository
- attributes: gets a local read-only copy of the attributes of a file in the centralized repository

The Think Team component sends messages *ttready*, *save*, *openfile* and *closefile* and receives the rest, while the CAD component sends *start, stop, setvalue, setvalues, attributes, remove, get, import, afterinit, checkin, checkout* and receives the other messages.

It is important to note that component Think Team has the further relevant scenarios of execution, which can be infinitely repeated in any order, from the initial state:

- initialize: $start?ttready!afterinit?$
- read file attributes: $attributes?openfile!closefile!$
- read file: $get?openfile!closefile!$
- import file: $import?openfile!save!closefile!$

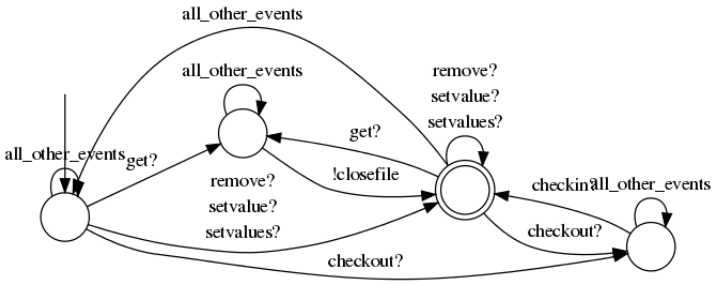
- lock file for writing: $checkout?openfile!$
- unlock file after writing: $checkin?save!closefile!$
- set a file attribute: $setvalue?$
- set all file attributes: $setvalues?$
- delete file: $remove?$
- shut down: $stop?$

In order to ensure the correctness and consistency of the information management process, the resulting composed system must conform to a specific coordination policy (see figure 5.25(a)). The original policy said: "A document cannot be removed if someone has checked it out. Moreover, the attributes cannot be modified if someone is getting a copy of the entity as a reference model." [78]. However, in order to make the property more specific, the rewritten coordination policy in figure 5.25 expresses the correct sequences in which a file can either be removed or edited: either directly from the initial state, or after a previous read scenario has finished, or after the scenario for unlocking the file for writing has been initiated.

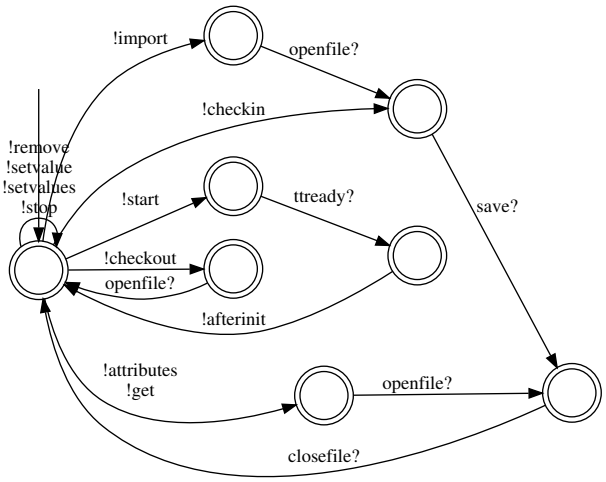
For controller synthesis, the Supremica tool [59] was again used. We assumed both ThinkTeam and CAD as incompletely specified, black box components. The ThinkTeam alphabet contains 15 events, out of which 4 are uncontrollable, while the alphabet of CAD has 11 uncontrollable events out of 15. The fairness bound θ thus becomes $\theta = 12$ for the Think Team component, and $\theta = 54$ for the CAD component. The bound on the longest acyclic path in the model is $m = 4$ for both Think Team and CAD. Although the value of m is relatively small, the larger alphabet and fairness bounds (especially in the case of the CAD component) have made the inferrence of complete models prohibitively hard for this case study, therefore we have only experimented with incomplete learning techniques.

In the performed experiments, we have studied the learning process of both Think Team and CAD entities in three situations. First, we learn each entity individually, without considering the safety property. Then, we experiment with the centralized learning process, using the required safety property. In the end, we also experiment with the distributed learning setting on this case study, making use again of the specified coordination policy.

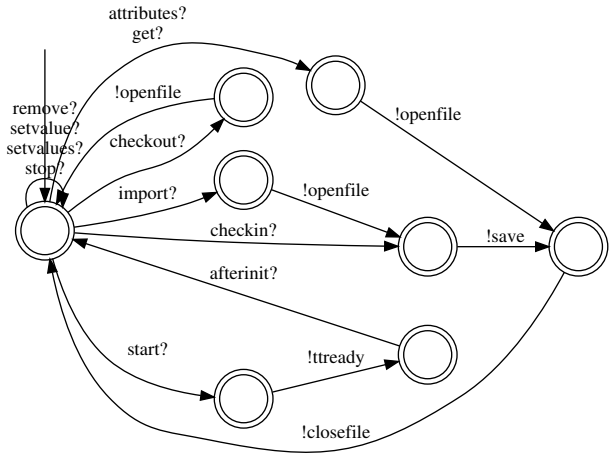
The cycle specifications files for all the three components in this case study are



(a)



(b)



(c)

Figure 5.25: Case study: 5.25(a) Property / Coordination policy, 5.25(b) CAD simplified model, 5.25(c) Think Team real model

included below. They are the same to be used during centralized, distributed and individual behaviour exploration.

The cycles specified for the optimized exploration of the Think Team component can be seen below. All considered cycles have only one join point, at the entry of each. Each such cycle contains one of the relevant execution scenarios of the Think Team component, earlier described and detailed.

```
:[1]<checkin?><save!><closefile!>
:[1]<import?><openfile!><save!><closefile!>
:[1]<checkout?><openfile!>
:[1]<get?><openfile!><closefile!>
:[1]<attributes?><openfile!><closefile!>
:[1]<start?><ttready!><afterinit?>
:[1]<remove?>
:[1]<setvalues?>
:[1]<setvalue?>
:[1]<stop?>
```

Below, one can also see the specified cycles for the cycle-based exploration of the CAD component. As the initial case study in [78] was synchronous and as it only has two protagonists, it is natural that in its asynchronous version the two participating components mirror each other, i.e. their behavioural model is similar under the consideration that a message sent event in one component becomes a (same) message receive event in the other. So, all relevant scenarios of the CAD component actually mirror the relevant scenarios described for the Think Team component. As the specified cycles for the CAD component also represent all the relevant use scenarios of CAD, it is easy to notice that, in fact, they perfectly mirror the cycles specified for the Think Team component.

```
:[1]<checkin!><save?><closefile?>
:[1]<import!><openfile?><save?><closefile?>
:[1]<checkout!><openfile?>
:[1]<get!><openfile?><closefile?>
:[1]<attributes!><openfile?><closefile?>
:[1]<start!><ttready?><afterinit!>
:[1]<remove!>
:[1]<setvalues!>
:[1]<setvalue!>
:[1]<stop!>
```

5.3.1 Individual Exploration

When studying the BASYL algorithm for model learning in an individual exploration setting, no safety property is not employed to coordinate the execution. In the performed experiments, the learning process for incomplete, approximate models is studied both with its original, unaltered exploration method and with the cycle-based optimization for behaviour exploration. In the latter case, the cycles used tp stir the exploration are the same specified cycles for use in the centralized and distributed exploration settings.

Think Team

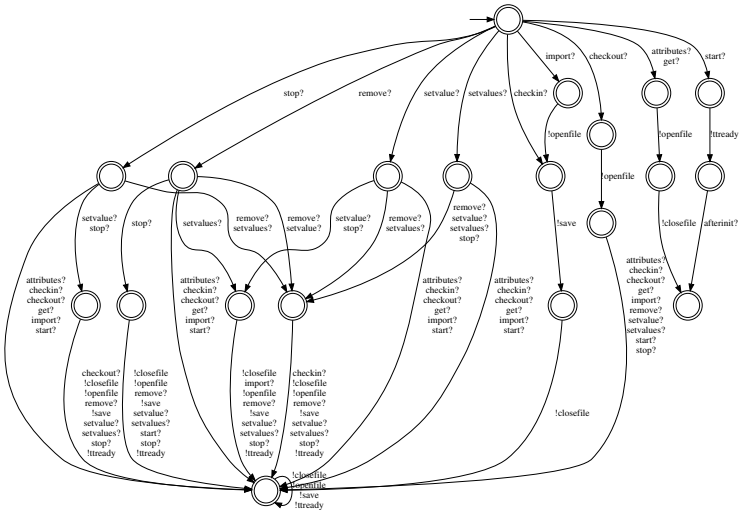
When learning the model of the Think Team component separately, the following values were set for the relevant learning parameters: m, the size of longest acyclic trace in the model, was set to $m = 4$, and θ, the fairness bound for the black box, was computed as $\theta = 12$.

Figures 5.26(a) and 5.26(b) show the models of the Think Team behaviour learned in a number of 500, respectively 1000 executions. In figure 5.26(a), almost only the states on the first two levels appear to be completely explored, while the rest still have uncontrollable transitions to the unknown-future state. No cycle has been identified in the model. The model in figure 5.26(b), obtained after 1000 executions, is somehow better explored than the previous one, as only 4 of its states still have uncontrollable transitions to the unknown-future state. However, still no cycle is identified.

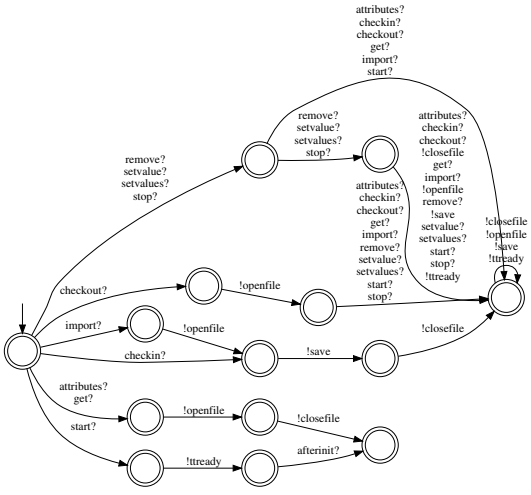
Further on, figure 5.27(a) shows us a model of Think Team learned in 1500 executions. The behaviour of the black box seems significantly better explored, although transitions to the "wild" future state still exist. Also, one of the model cycles has been identified. Then, in figure 5.27(b), after 15000, the model is explored well enough for 5 cycles to be identified in the model, although not from the initial state, as they should have been.

In figure 5.28(a), we can see a model of the Think Team component learned in 150000 executions. All the cycles from the precise model are present, although some appear more than once, and others were not identified from the initial state, as they originally are – this happens because it is easier for deeper states to be identified as identical. One of the states in the model still has uncontrollable transitions to the unknown-future state.

Finally, after 1500000 executions we obtain a precise-like model, that can be seen in figure 5.28(b). It is worth mentioning here that precise-like models have not been obtained neither for 250000, nor for 500000, and nor for 750000 runs, although all these experiments have produced models that contained all the original cycles and no longer had transitions to the unknown-future state. However, several states in these models were yet to be identified as identical.

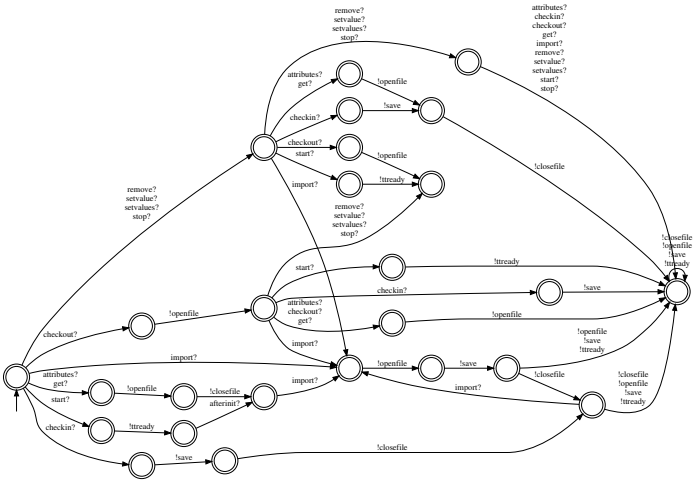


(a)

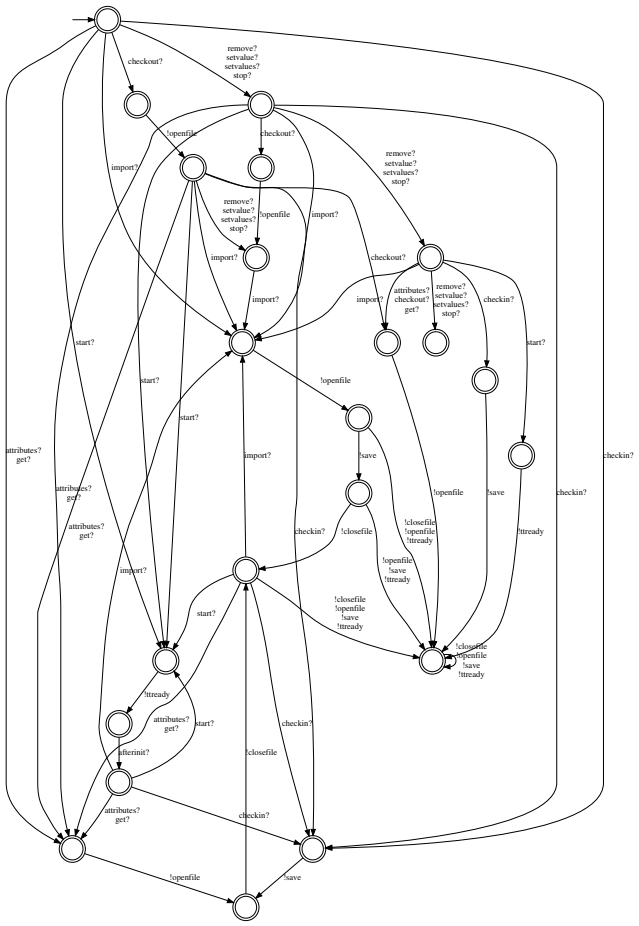


(b)

Figure 5.26: Learned models for component Think Team in: (a) 500 executions, (b) 1000 executions.

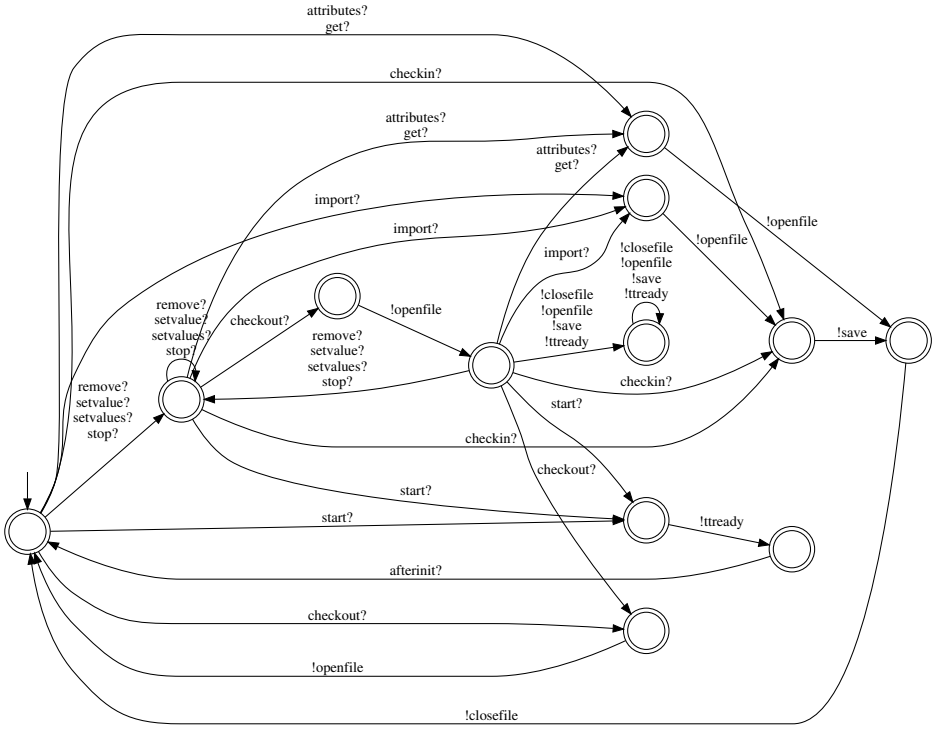


(a)

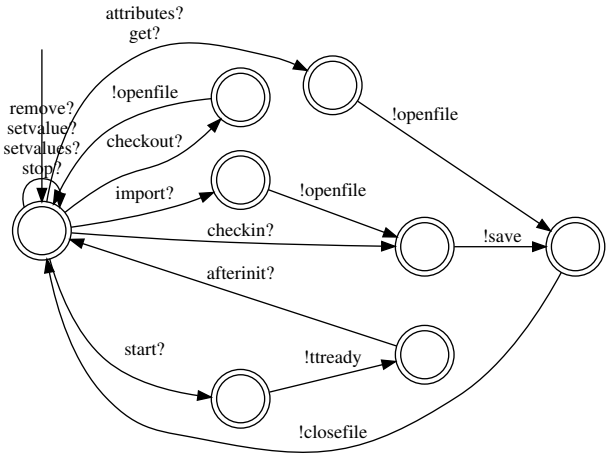


(b)

Figure 5.27: Learned models for component Think Team in: (a) 1500 executions, (b) 15 000 executions.



(a)



(b)

Figure 5.28: Learned models for component Think Team in: (a) 150 000 executions, (b) 1500 000 executions.

Think Team – cycle-based exploration

The CAD and Think Team case study is one where the benefits brought by cycle-based exploration over the original, unoptimized exploration strategy are highly visible. The relevant cycles are identified faster, the models obtained, even when far from precise-like, have a smaller number of states, as the result of a more focused exploration.

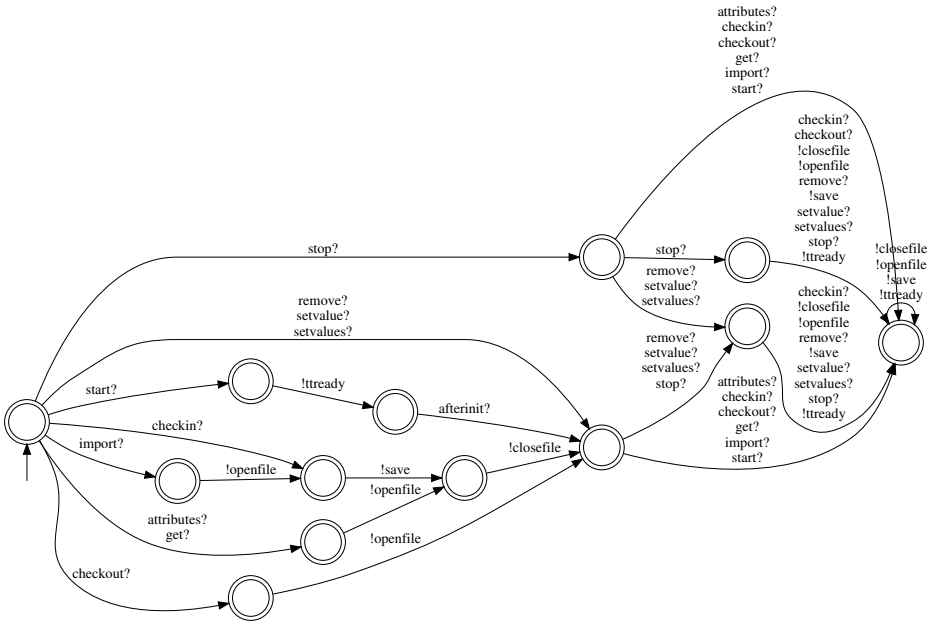
The specification of all the cycles considered relevant for the behaviour exploration process is given again below.

```
:[1]<checkin?><save!><closefile!>
:[1]<import?><openfile!><save!><closefile!>
:[1]<checkout?><openfile!>
:[1]<get?><openfile!><closefile!>
:[1]<attributes?><openfile!><closefile!>
:[1]<start?><ttready!><afterinit?>
:[1]<remove?>
:[1]<setvalues?>
:[1]<setvalue?>
:[1]<stop?>
```

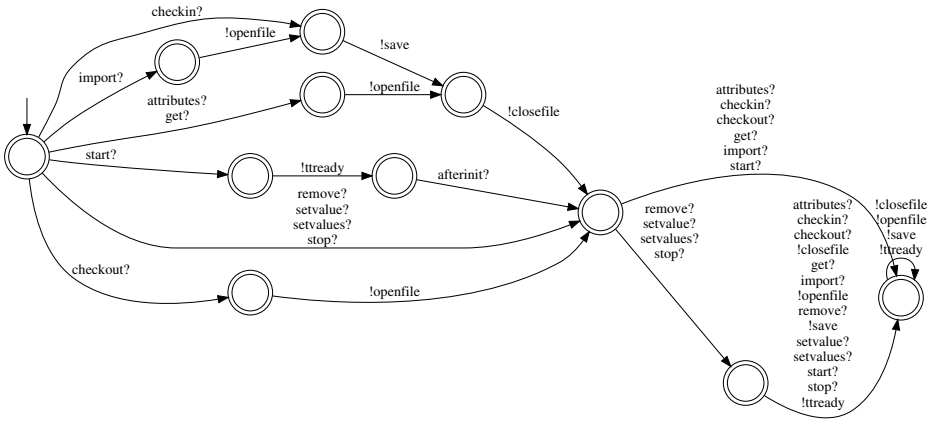
In figures 5.29(a) and 5.29(b) we can see models of the Think Team component obtained after 500, respectively 1000 executions. Although none of the relevant cycles have been identified yet, the models obtained have a smaller number of states and less transitions to the unknown-future state than the models obtained by unoptimized learning. The model at figure 5.30, obtained after 1500 executions, has still one identified cycle, similarly to its correspondent resulting from regular learning, but shows a greater number of completely explored states, and at greater depths. At figure 5.31 we see a model obtained in 15000 executions, that still has only 5 identified cycles, like its correspondent from unoptimized exploration, but also having more completely explored states at significant depths.

However, after no more than 150000 executions, the cycle-based learning algorithm returns a precise-like model as a result. It is important to emphasize here that the Think Team component has a rather large alphabet, of 15 events, and a fairness bound of 12. This makes it more difficult to explore it completely than would be the case with simpler components. In order to have completely explored states at depth 2, for example, an actual number of executions that

surpasses $12^2 = 144$ is needed. At depth 3, we would need more than 12^3, etc. What the cycle-oriented exploration actually does is to obtain more than θ visits for deeper vertices on interesting paths in a faster way, due to acutely stressing those paths by a focused exploration process.

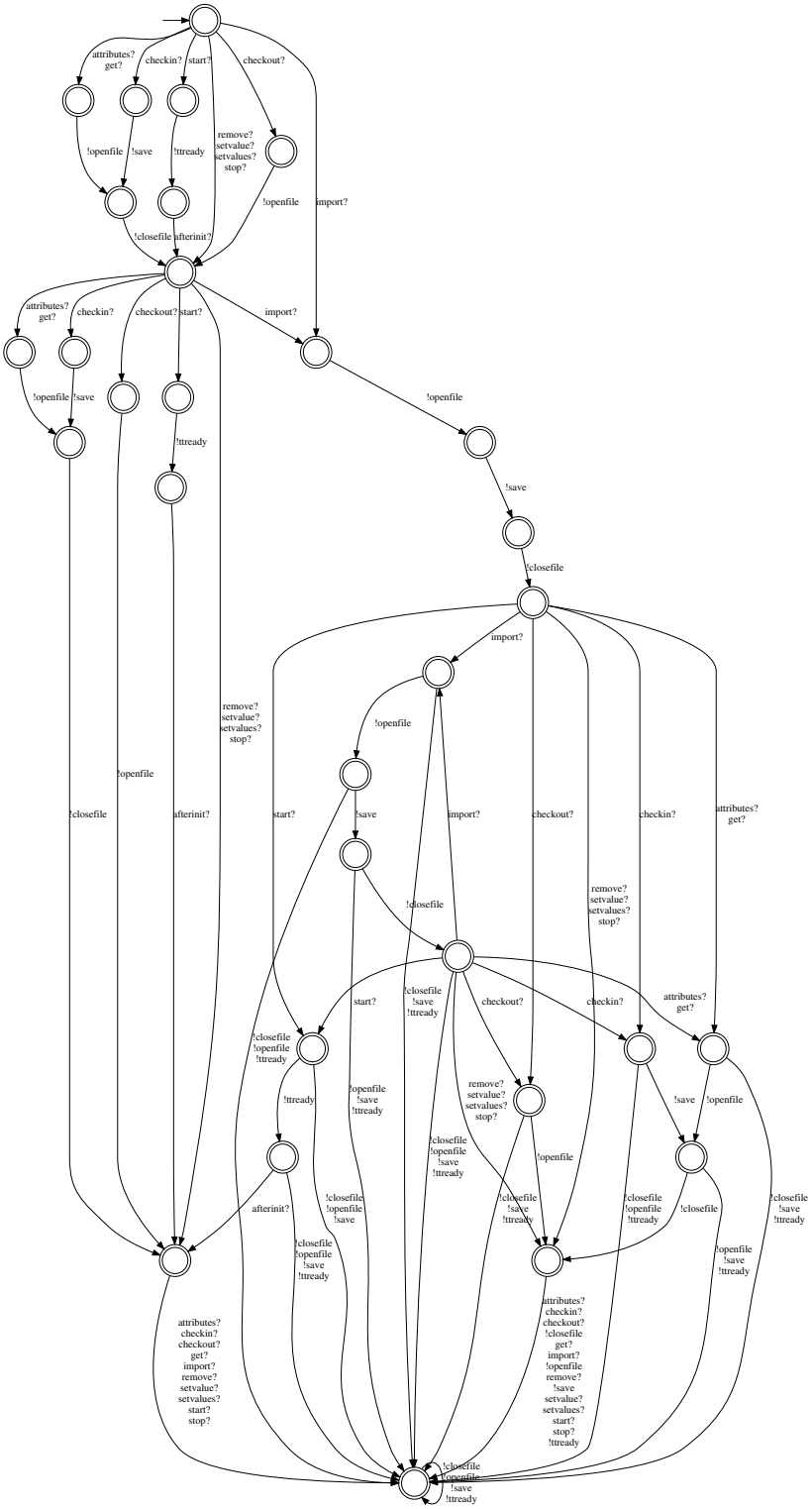


(a)



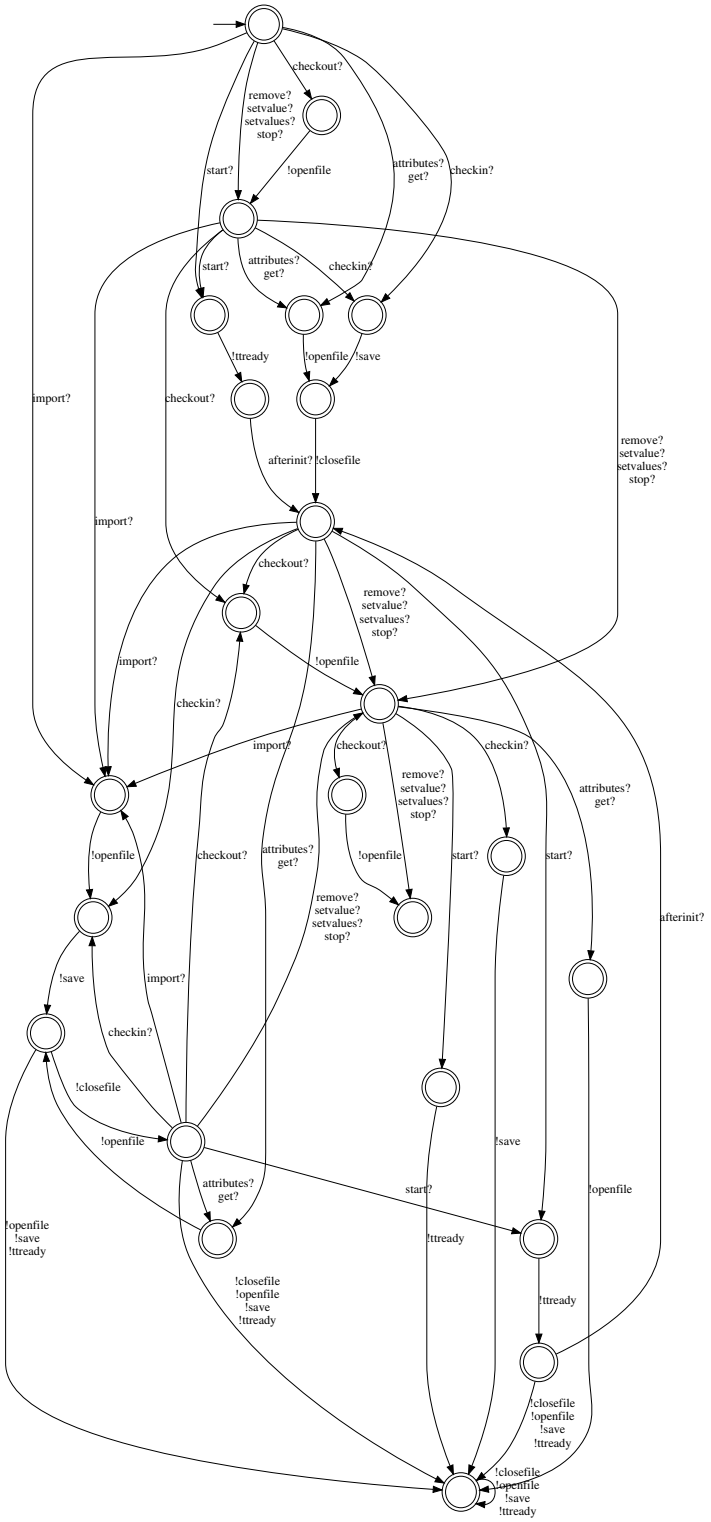
(b)

Figure 5.29: Learned models for component Think Team in: (a) 500 executions, (b) 1000 executions.



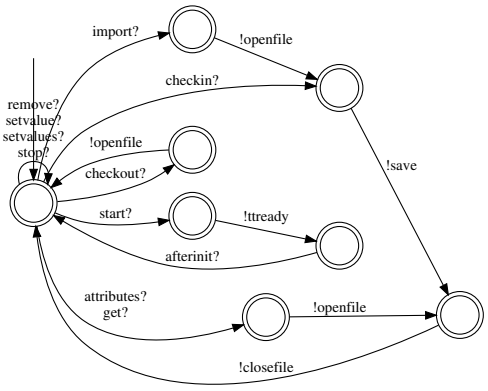
(a)

Figure 5.30: Learned model for component Think Team in: (a) 1500 runs.



(a)

Figure 5.31: Learned model for component Think Team in: (a) 15 000 runs.



(a)

Figure 5.32: Learned model for component Think Team in: (a) 150 000 runs.

CAD

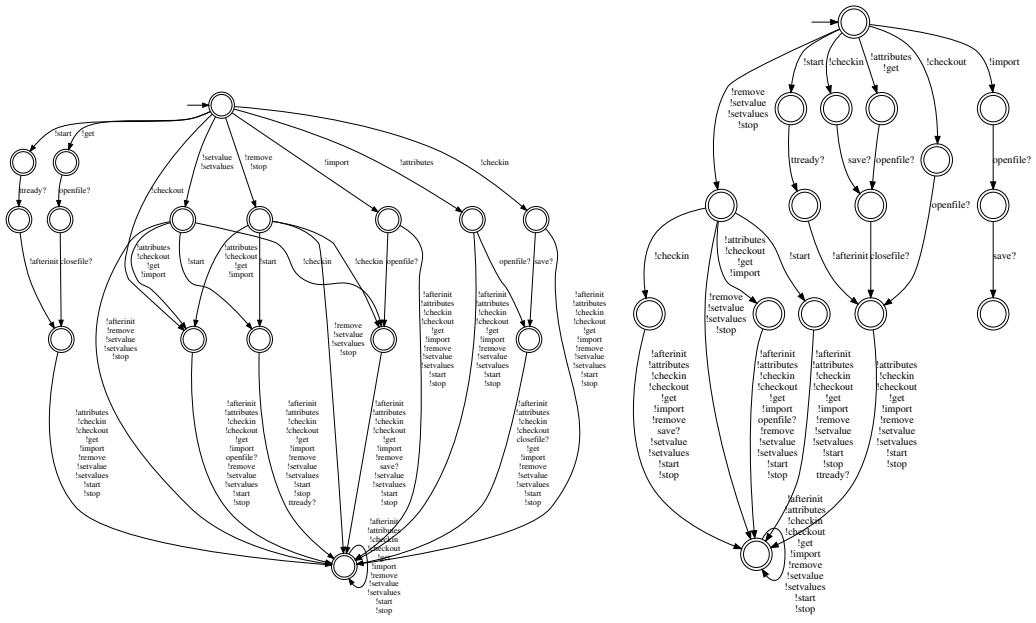
In the case of the CAD component, the further value were attributed to the relevant learning parameters m, the size of longest acyclic trace in the model, was set to $m = 4$, and θ, the fairness bound, was computed as $\theta = 54$. We have to specifically observe that CAD has a significantly large number of uncontrollable events in its alphabet: 11 out of 15. This not only leads to a high fairness bound, but also makes the black box component CAD harder to control, and, consequently, harder to learn.

In figures 5.33 and 5.34, we can see the models resulting from a lower set bound on the maximum number of executions. Thus, in figure 5.33(a) we see a model of CAD learned after 500 executions, in figure 5.33(b) we see a model learned in 1000 executions, while the model in figure 5.34(a) has been learned in no more than 5000 executions. While the first two models appear to offer few identified paths, as a result of a poor exploration, the latter model shows a better exploration of potential paths, and also a more clean model, since at least the states on the first two levels are now completely explored. However, many states have uncontrollable transitions to the unknown-future state, and no cycle is yet identified.

Further on, we consider models obtained after a considerably larger number of executions. In figure 5.35(a) we can see the result of a learning process that uses 50000 runs. The model obtained is significantly larger, as much more potential paths have been explored, however still no cycle has been identified

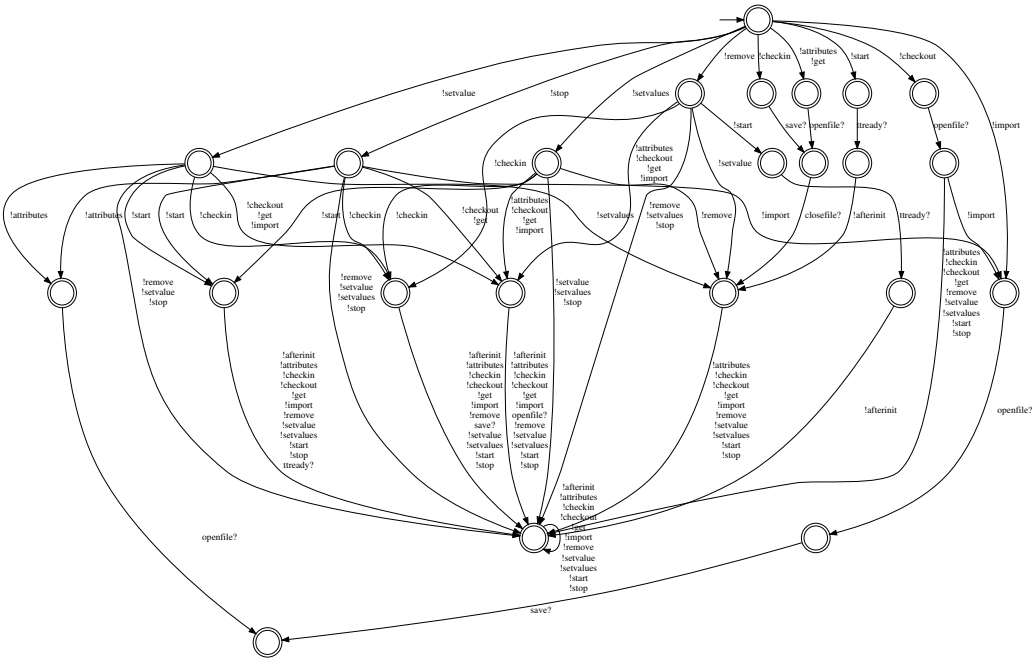
and there are still a large number of states which have uncontrollable transitions to the unknown-future state. The model in figure 5.36(a) has been obtained after 250000 learning executions. Finally, the states on the third level of depth have been completely explored and 4 of the cycles in the original model have been identified. The number of states with uncontrollable transitions to the unknown-future state has decreased, but is still substantial.

After 500000 executions, the learned model at figure 5.37(a) has fewer incompletely explored states, however the number of identified cycles remains 4. Finally, after 1500000 executions, the model at figure 5.37(b) is obtained, which has all the original 10 cycles identified, and 5 of them are actually identified from the initial state. This model still has a number of 5 states exhibiting uncontrollable transitions to the unknown-future state, however, this is normal, as $\theta^4 = 8503056$, which highly surpasses our imposed limit of 1500000 runs.



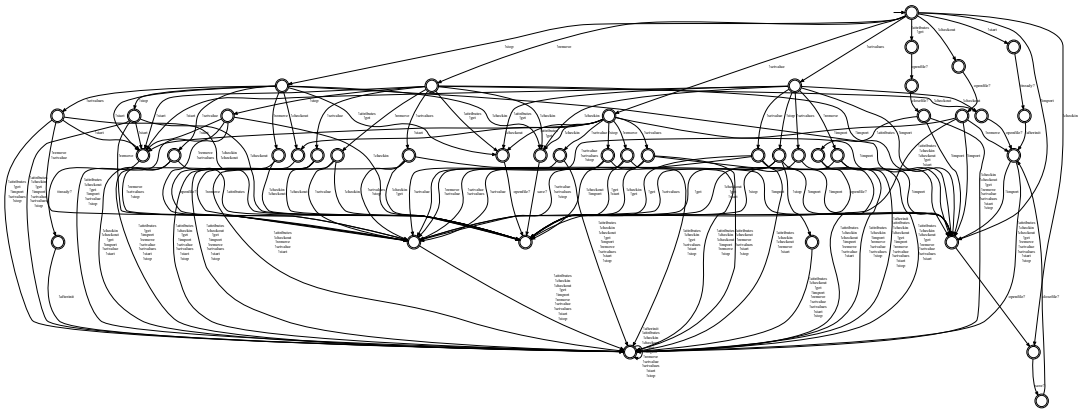
(a) (b)

Figure 5.33: Learned models for component CAD in: (a) 500 executions, (b) 1000 executions.



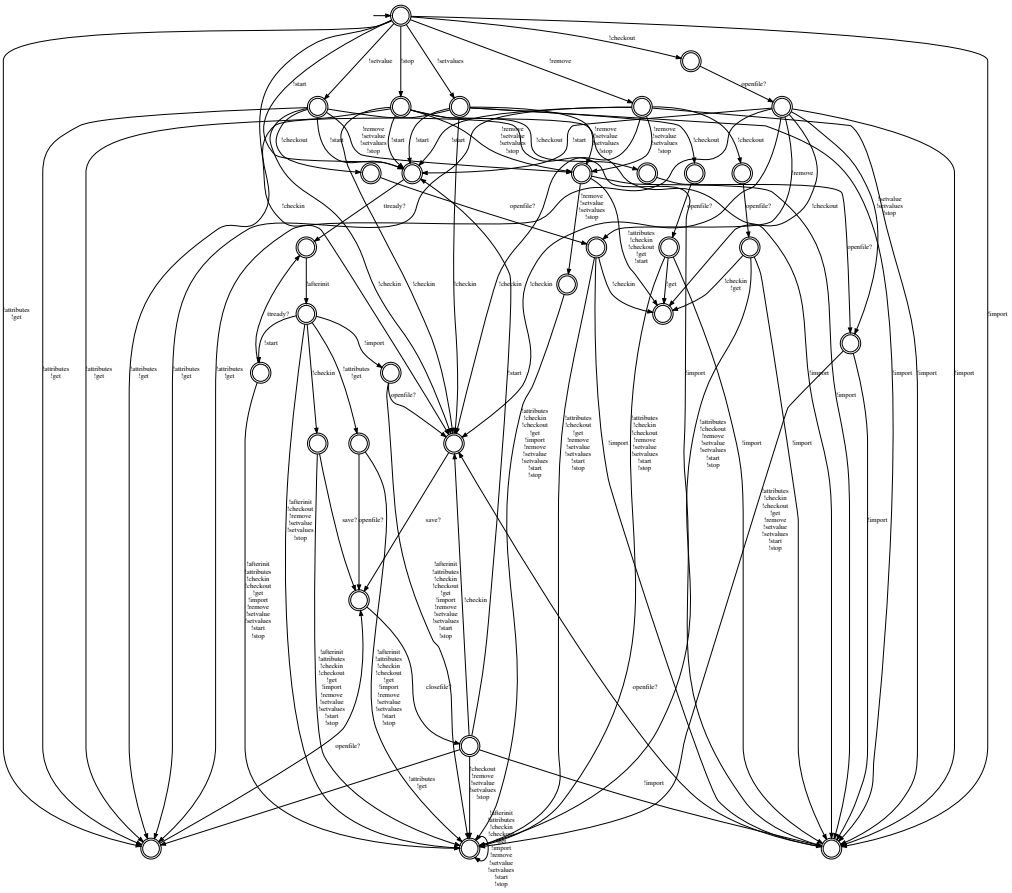
(a)

Figure 5.34: Learned models for component CAD in: (a) 5000 executions.



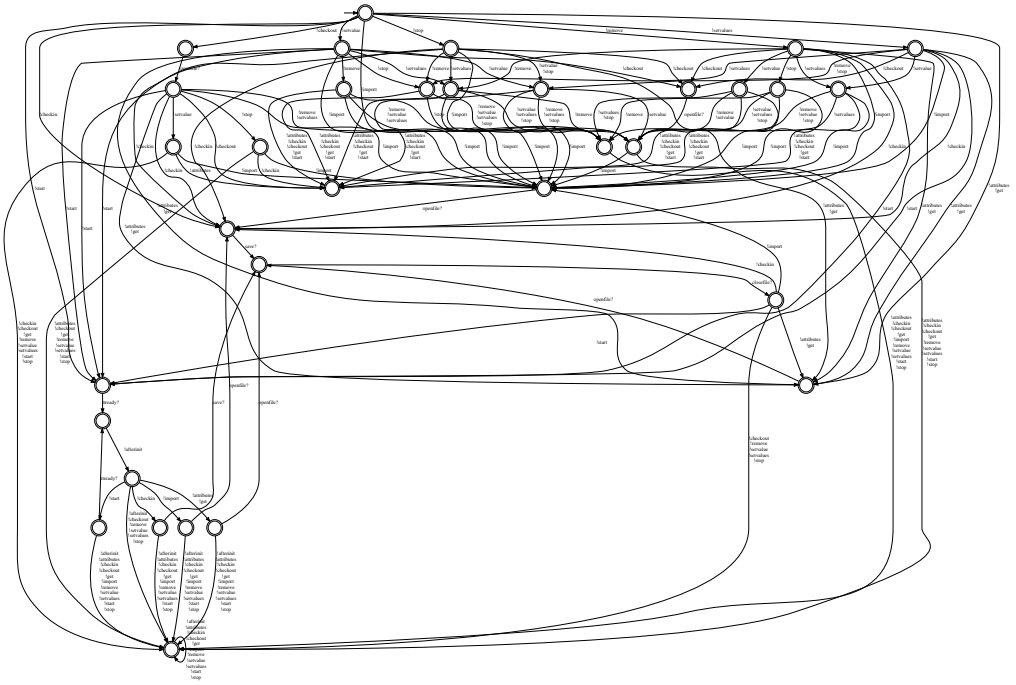
(a)

Figure 5.35: Learned models for component CAD in: (a) 50 000 executions.

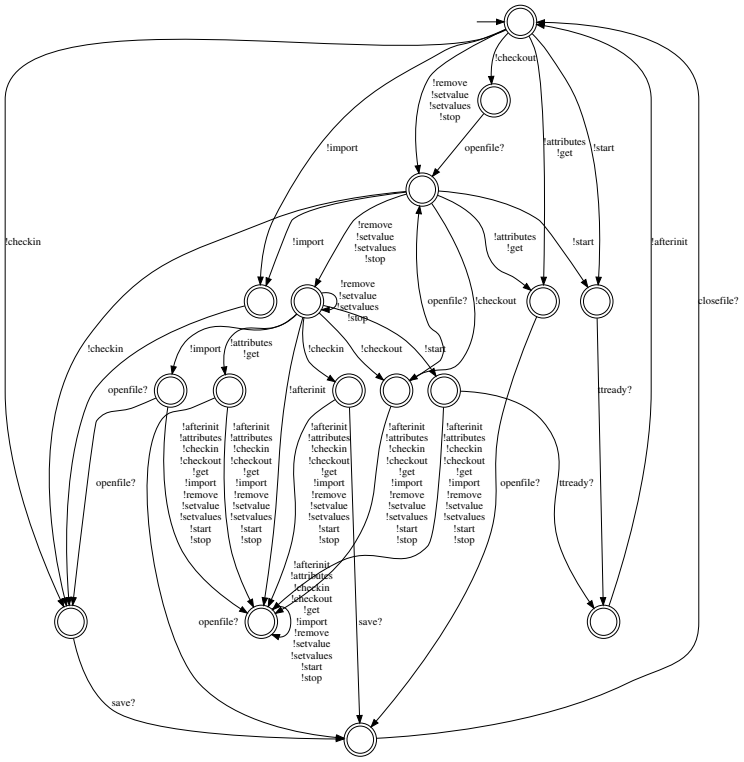


(a)

Figure 5.36: Learned models for component CAD in: (a) 250 000 executions.



(a)



(b)

Figure 5.37: Learned models for component CAD in: (a) 500 000 executions, (b) 1500 000 executions.

CAD – cycle-based exploration

As the CAD component has a large number of uncontrollable events in its alphabet, being thus hard to control, the influence the cycle-based optimized exploration has on the learning process is significantly less than the one it had in the case of the Think Team component. Steering a current execution towards one scenario or another relies on the controllable events in the alphabet, which are only in small percentage here.

The specified cycles for the CAD black box are given below.

```
:[1]<checkin!><save?><closefile?>
:[1]<import!><openfile?><save?><closefile?>
:[1]<checkout!><openfile?>
:[1]<get!><openfile?><closefile?>
:[1]<attributes!><openfile?><closefile?>
:[1]<start!><ttready?><afterinit!>
:[1]<remove!>
:[1]<setvalues!>
:[1]<setvalue!>
:[1]<stop!>
```

In figures 5.39(a), 5.38(b) and 5.38(c) we have the learned models for component CAD after 500, 1000 and respectively 5000 executions. They are pretty similar to the ones obtained for the unoptimized learning algorithm and have same main characteristics: states at small depths are incompletely explored, no cycles are identified, and the first two models are proof of poor exploration.

However, further on slight differences arise. In figure 5.39(a), the model obtained after 50000 executions actually has one identified cycle, instead of none, as it is the case with its correspondent in the unoptimized setting. Also, in figure 5.39(b), for the model learned in 250000 executions, the identified cycles are in number of 5, instead of 4, as obtained with the regular learning algorithm.

After 500000 runs, the model in figure 5.40(a) is obtained, which still has 5 identified cycles, but shows significantly less states linked to the unknown-future state. And, finally, after 1500000 learning executions, we obtain the model in figure 5.40(b), which is highly similar to the one obtained by the unoptimized algorithm. Thus, as CAD is far less controllable than Think Team, the effect of the cycle-based optimization is only marginally visible here, and it appears only where controllable choices exist.

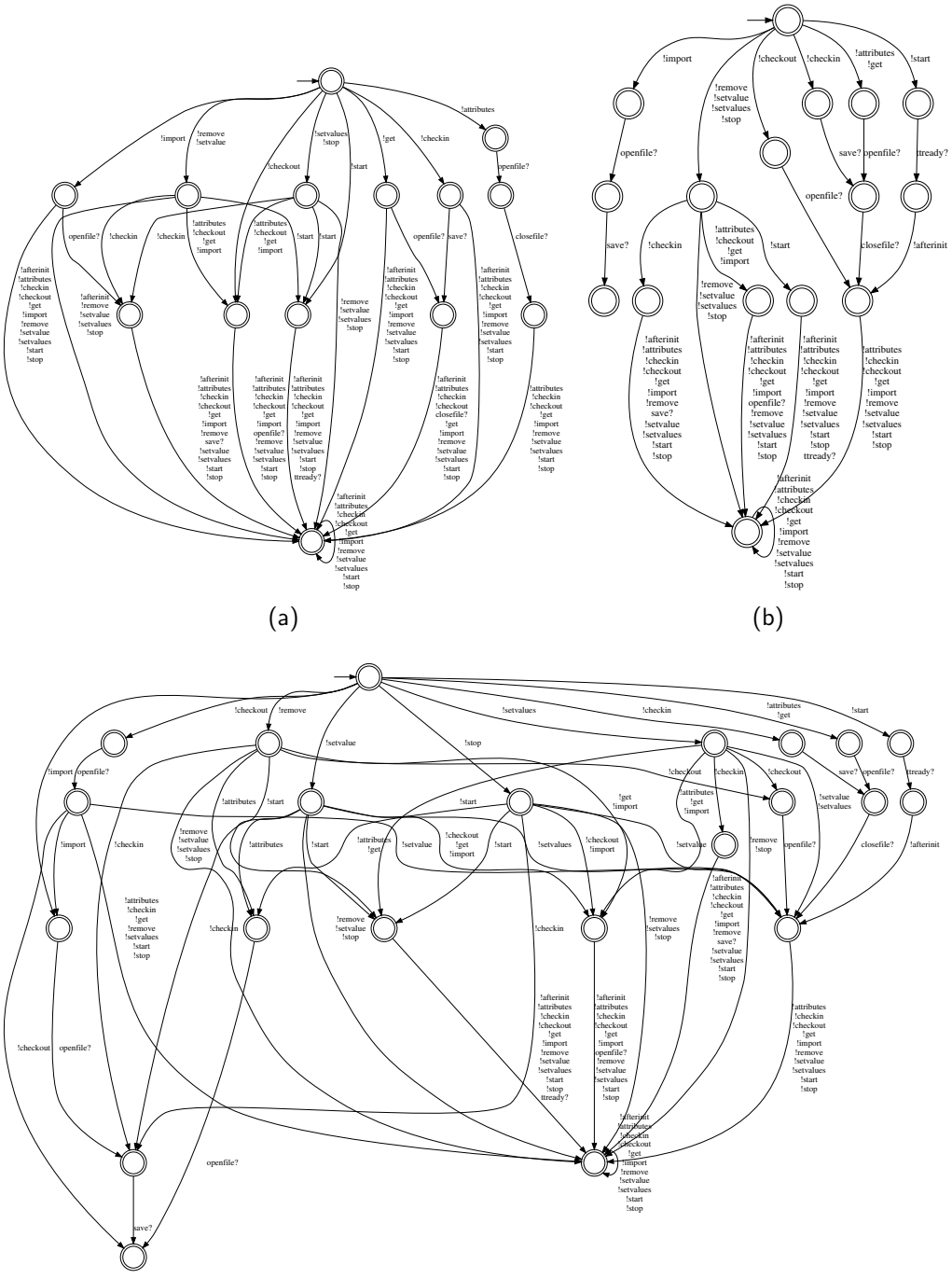
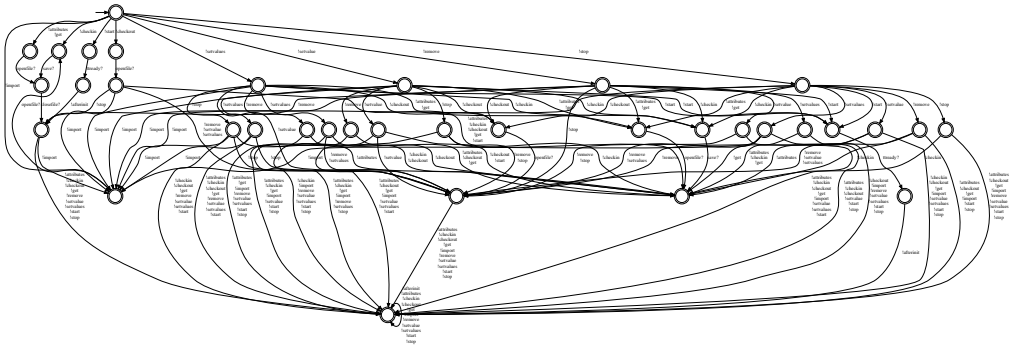
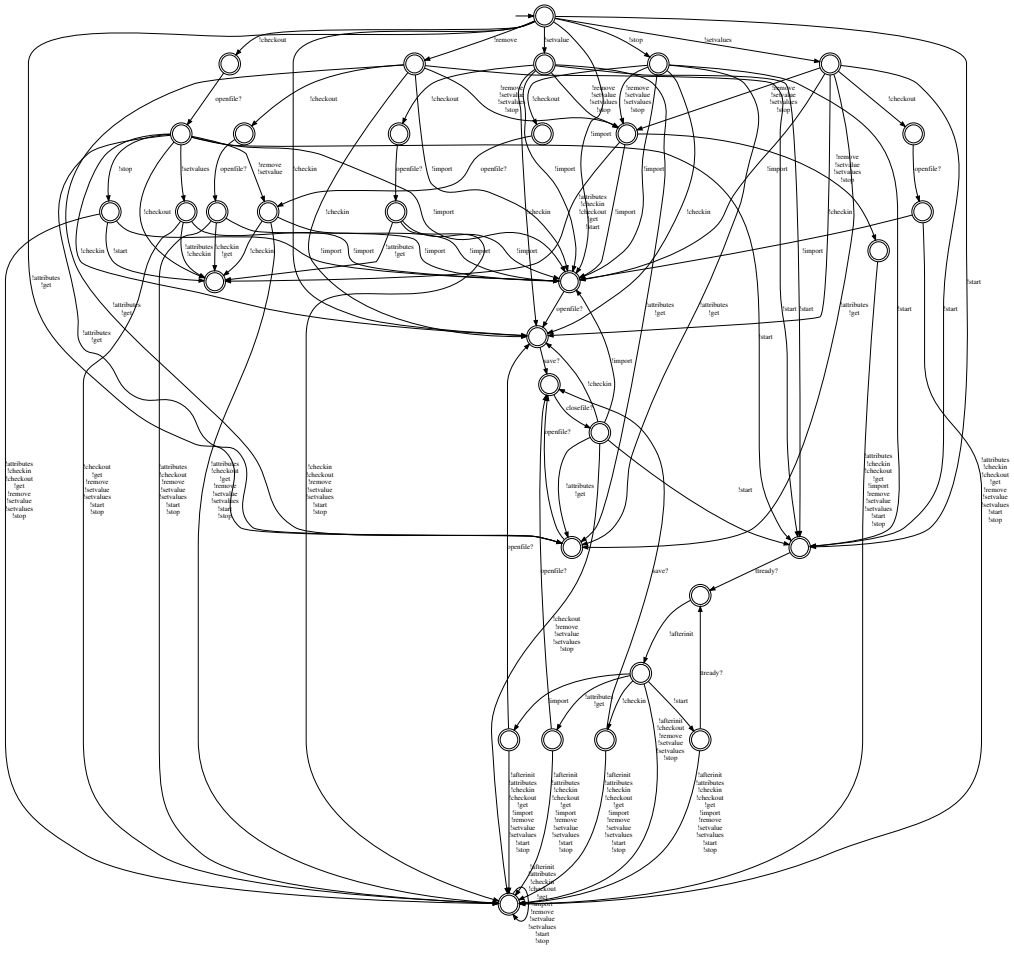


Figure 5.38: Learned models for component CAD in: (a) 500 executions, (b) 1000 executions, (c) 5000 executions.

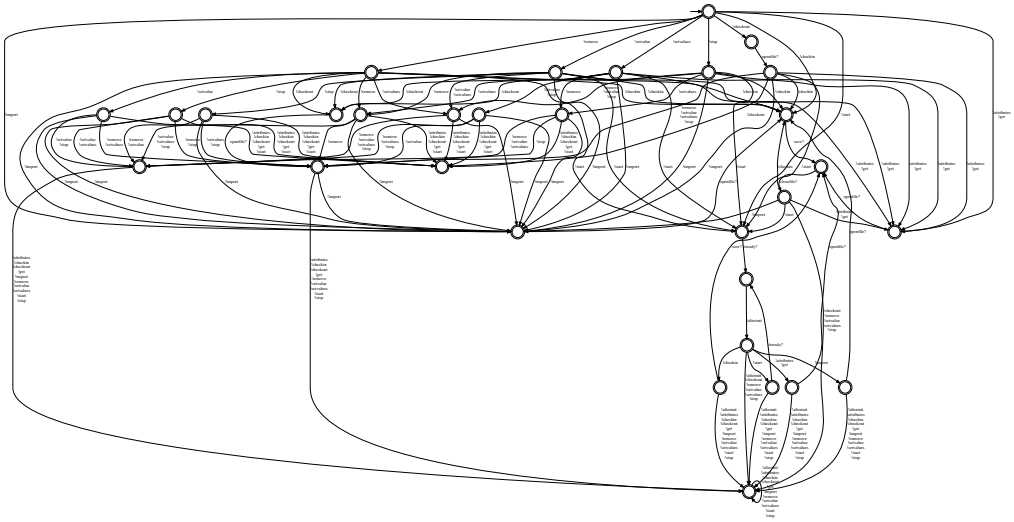


(a)

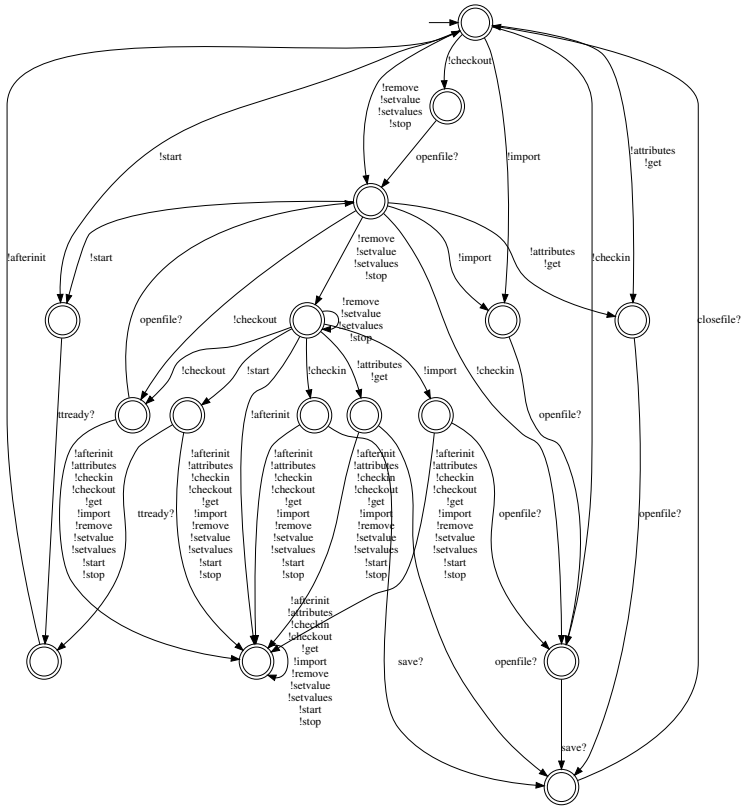


(b)

Figure 5.39: Learned models for component CAD in: (a) 50 000 executions, (b) 250 000 executions.



(a)



(b)

Figure 5.40: Learned models for component CAD in: (a) 500 000 executions, (b) 1500 000 executions.

5.3.2 Centralized Exploration

In the following the model inference results obtained for the case study of Think Team and CAD, using a centralized exploration setting, are presented. As this case study considers a system with only two components, the global traces used by the centralized learning process, while still longer than the sum of traces used in the distributed learning setting, do not exhibit significantly greater lengths. However, having longer execution traces is a characteristic of centralized learning that does not depend on any specifics of the case study.

Learning experiments performed on the Product Data Management System case study have used global size for the longest acyclic path $m = 4$. The fairness bounds were again set independently for each component, thus $\theta = 12$ for the Think Team component and $\theta = 54$ for the CAD component. After each learning experiment, a controller has been computed for the inferred models and the coordination policy. We have then verified whether the obtained controller did work on the real system, i.e. on the plant determined by the precise models of participating components. For controller synthesis and verification we have used the Supremica tool [59].

Normal Exploration

Below we present the results of learning the models of the Think Team and CAD components using the centralized exploration algorithm. Table 5.5 summarizes the statistical data characterizing these results, i.e. the inferred models and the generated system controllers. The data presented represents the number of states and the number of transitions of both automata and the controller.

Some of the models obtained from the performed learning experiments can be seen in figures 5.41 and 5.42 for the Think Team component, and in figures 5.43, 5.44 and 5.45 for the CAD component. After 500 and 1000 executions, the models in figures 5.41(a), respectively 5.41(b) were obtained for Think Team, while models in figures 5.43(a), respectively 5.43(b) were learned for CAD. As no cycle is identified in any of these models, the generated controllers would only allow an execution scenario to happen at most once.

In figure 5.41(c) we can see a model of Think Team, and in figure 5.43(c) one of CAD, both learned in 10000 executions. Both models have the same identified cycle, thus the computed controller with 324 states and 1982 allows the import

file scenario an unlimited number of repetitions, while all other scenarios can repeat only once. This points to an advantage of the centralized exploration over the distributed and isolated settings: an identified cycle is more likely to appear in all participating components, since this implies that the scenario was explored globally for a significant number of times. However, when fairness bounds differ, cycle identification in the components with higher fairness bounds is clearly delayed. This is the case with the results obtained after 25000 executions: the model of the Think Team component can be seen in figure 5.42(a), while the one of CAD appears in figure 5.44(a). While the learned model of Think Team already has 5 identified cycles, the one for CAD has only one, so the obtained controller for the system is no more permissive than the previous one.

After 100000 executions, the models obtained have both the same 5 identified cycles (see figure 5.42(b) for Think Team and figure 5.44(b) for CAD), thus their corresponding controller, having 419 states and 4000 transitions, will also allow for these scenarios to repeat an unlimited number of times.

The models obtained in 500000 learning executions can be seen in figures 5.42(c) (Think Team), and 5.44(c) (CAD). The learned model for Think Team has all its cycles identified, 6 of them from the initial state, however the model obtained for CAD has only 5 identified cycles, so only the scenarios associated with these cycles will be allowed to repeat for an unlimited number of times by the computed 280 state and 2771 transition controller.

After 750000 learning executions, the model obtained for Think Team is similar to the one at figure 5.42(c) however, the model of the CAD component, which can be seen at figure 5.45(a), has a number of 7 identified cycles. Thus, the resulting controller, which has 280 states and 3003 transitions will allow for 7 out of 10 execution scenarios to repeat for an unlimited number of times.

The average trace length encountered during the normal exploration process was 19.16, with 3.16 more than 16, the sum of the two maximum trace lengths. It is worth mentioning here that the experiment using 1500000 executions has failed due to an Out of Memory exception thrown by the Java Virtual Machine.

Table 5.5: Model inference results – normal exploration

nr.	TT size	TT tr.	CAD size	CAD tr.	ctrl size	ctrl tr.
500	177 21	772 109	163 11	1825 126	133	1886
1×10^3	255 14	1172 56	194 16	2027 85	129	1048
1.5×10^3	250 26	925 71	221 16	2324 85	129	1048
2.5×10^3	250 26	925 71	245 16	2588 85	129	1048
5×10^3	249 26	925 71	254 24	2547 138	193	1656
1×10^4	226 23	927 73	250 19	2150 73	324	1982
2.5×10^4	206 22	373 78	239 22	2148 96	331	2584
5×10^4	204 29	378 106	243 38	1767 254	571	5786
1×10^5	121 20	275 69	193 22	1024 148	419	4000
2.5×10^5	114 12	325 46	201 27	1027 189	200	1801
5×10^5	114 12	325 46	206 37	1024 289	280	2771
7.5×10^5	113 12	324 46	128 37	993 329	280	3003

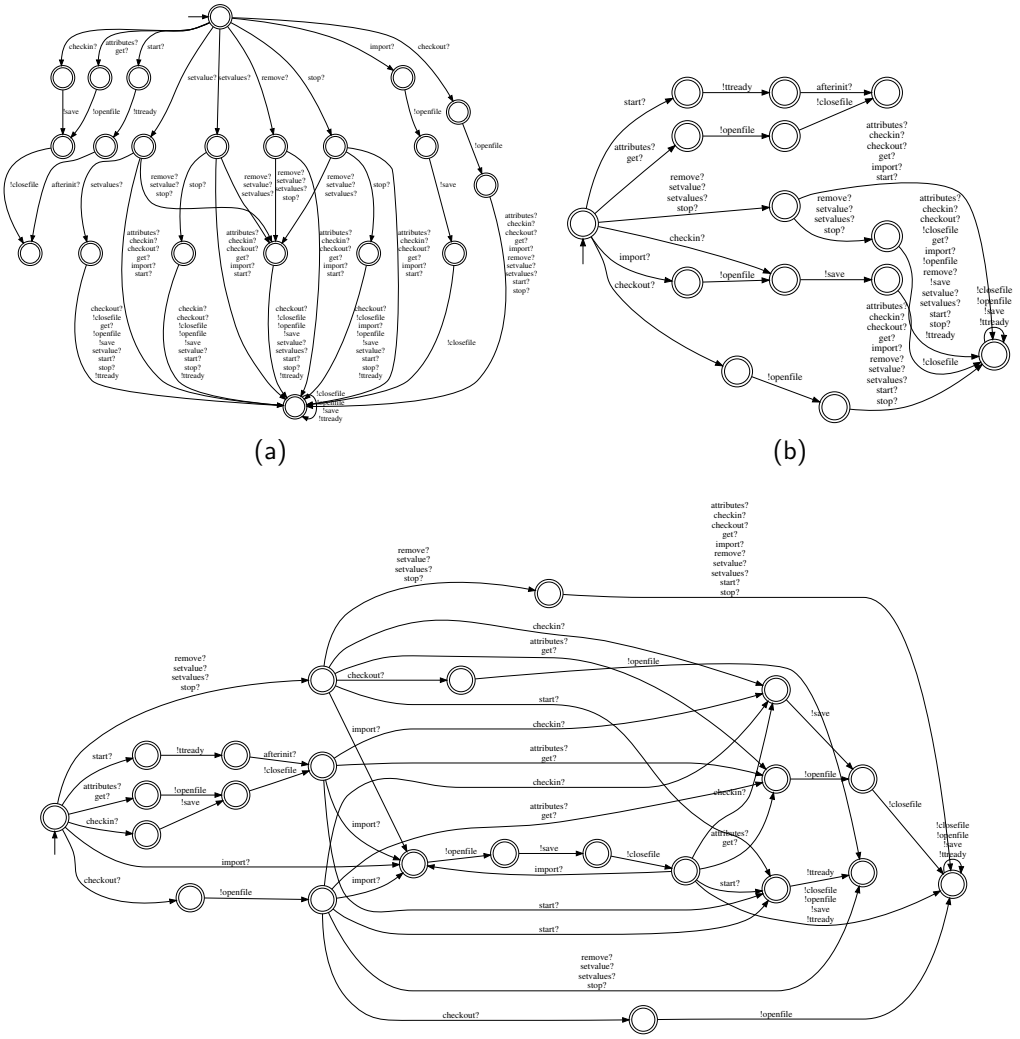
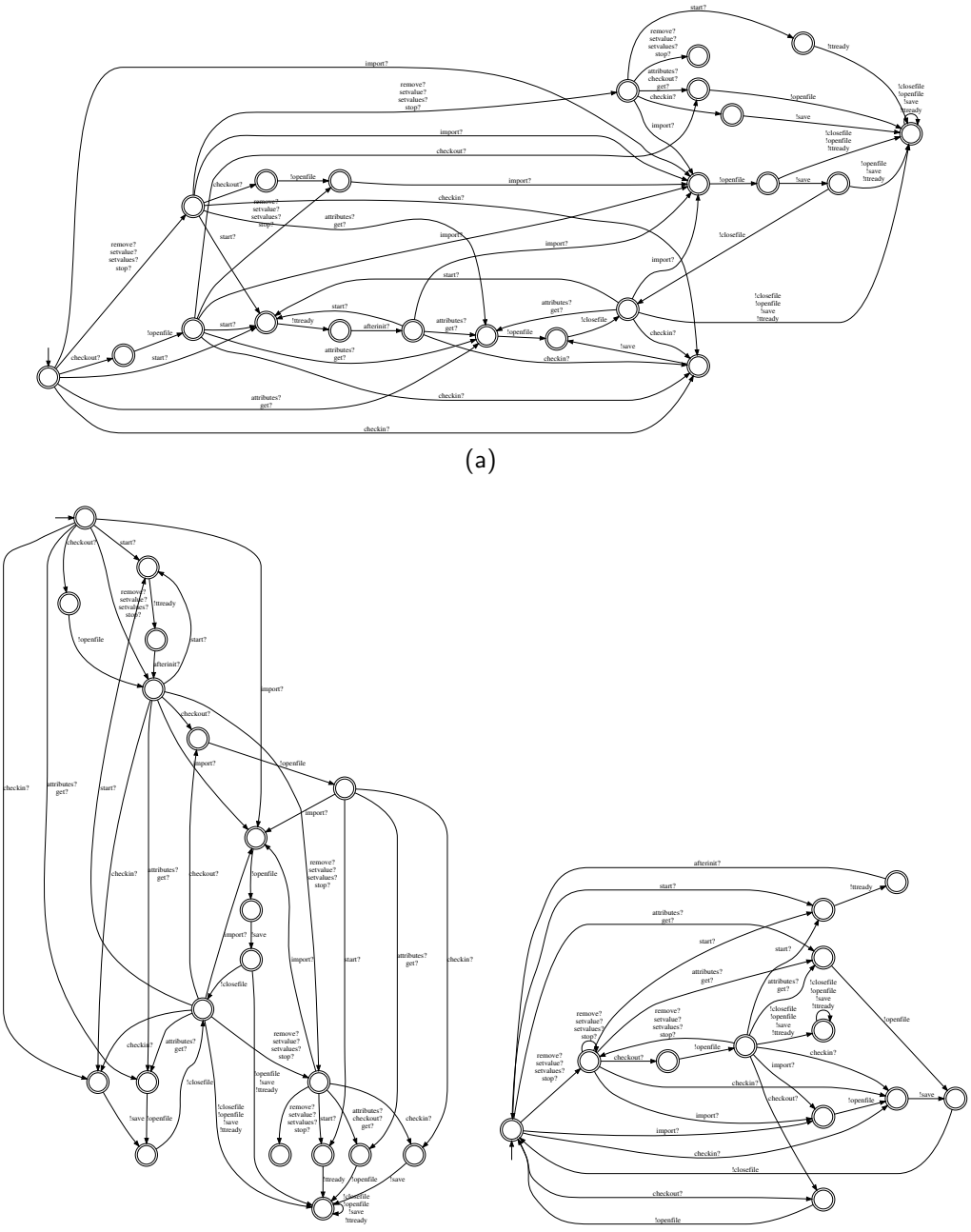


Figure 5.41: Learned models for component Think Team in: (a) 500 executions, (b) 1000 executions, (c) 10000 executions.



(a)

(b)

(c)

Figure 5.42: Learned models for component Think Team in: (a) 25 000 executions, (b) 100 000 executions, (c) 500 000 executions.

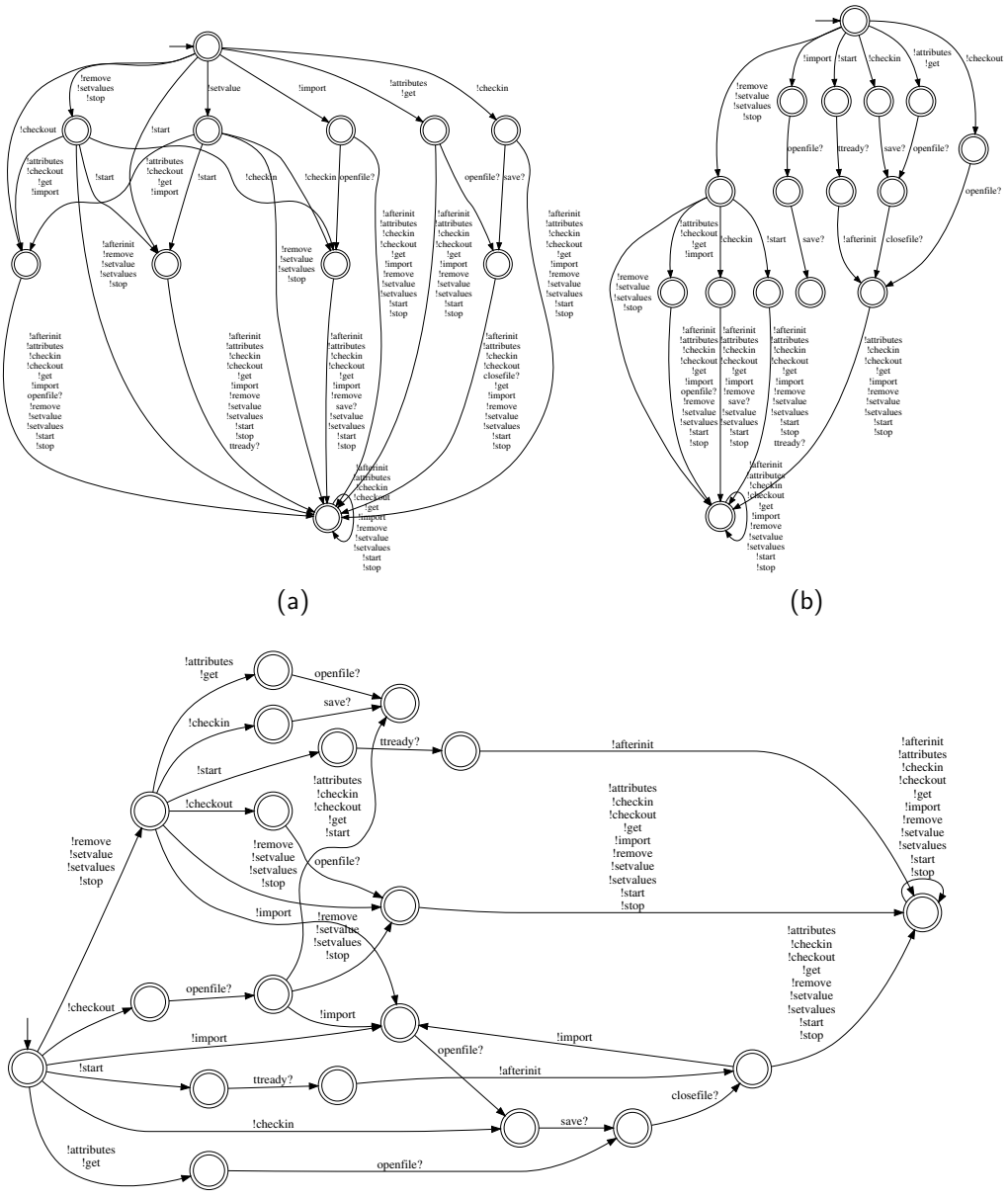
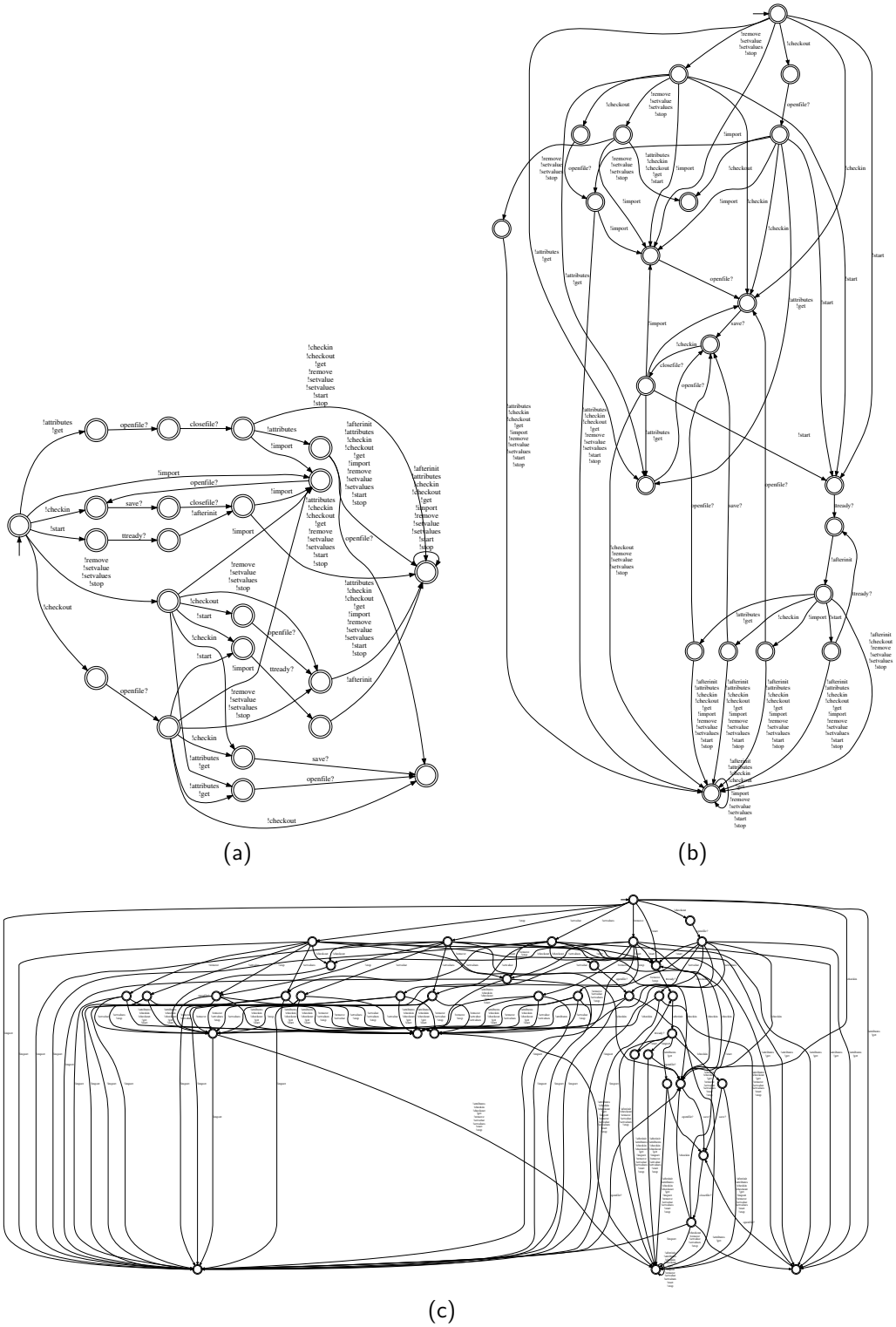


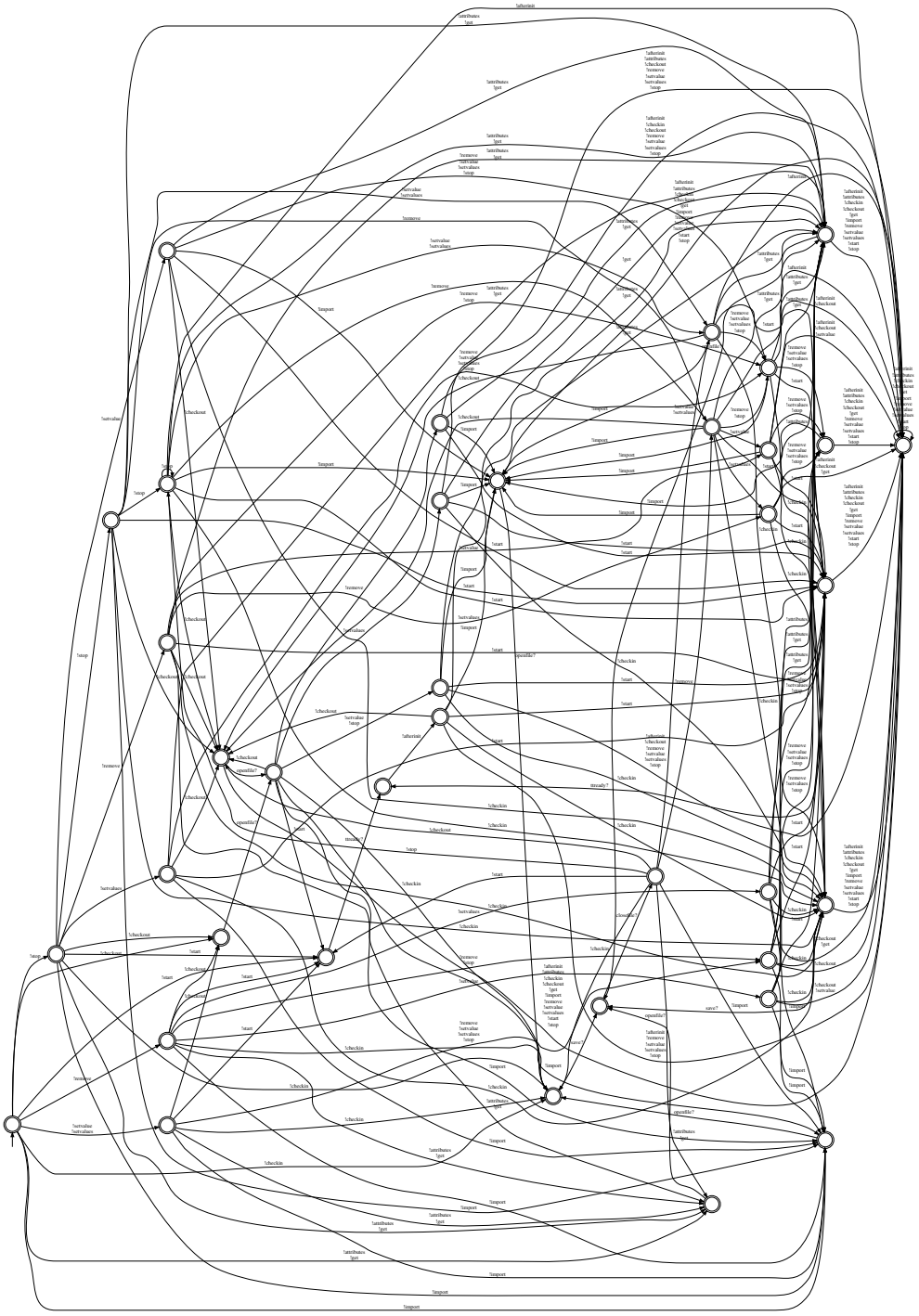
Figure 5.43: Learned models for component CAD in: (a) 500 executions, (b) 1000 executions, (c) 10 000 executions.



(a) (b)

(c)

Figure 5.44: Learned models for component CAD in: (a) 25 000 executions, (b) 100 000 executions, (c) 500 000 executions.



(a)

Figure 5.45: Learned models for component CAD in: (a) 750 000 executions.

Cycle-based Exploration

The cycle-based exploration interestingly improves the results of model learning in the centralized exploration setting for this test case. The improvement is asymmetrical, providing a substantial benefit, as expected, only for the highly controllable component Think Team, while not being of much use on the significantly harder to control CAD component.

Table 5.6 summarizes the cycle-based learning process by presenting relevant statistical data about the results, i.e. the inferred models and the generated system controllers. The data presented represents the number of states and the number of transitions of both automata and the controller.

The results of the cycle-based learning algorithm are further on commented from a qualitative point of view. After 500 and 1000 executions, the models in figures 5.46(a), respectively 5.46(b) were obtained for Think Team, while models in figures 5.50(a), respectively 5.50(b) were learned for CAD. Similarly to the results obtained with unoptimized exploration, the models seem poorly explored, no cycle is yet identified and the generated controllers would only allow an execution scenario to happen at most once.

In figures 5.47(a) and 5.50(c) the models of Think Team, respectively CAD, learned together in 10000 executions, can be seen. Just as their correspondents from unoptimized learning, both models have the same identified cycle. The controller obtained for this case has 533 states and 3428 and allows the import file scenario to repeat for an unlimited number of times, while the other scenarios can repeat only once. Again, just as in the unoptimized setting, after 25000 executions the model of the Think Team component already has 5 identified cycles (see figure 5.47(b)), while the one of CAD has just one cycle (figure 5.51(a)). The obtained controller for the system, having 463 states and 3297 transitions, does not actually allow for more repeating scenarios that the controller obtained for the previous case.

Similarly to the unoptimized case, after 100000 executions, the models obtained have both the same 5 identified cycles (see figure 5.48(a) for Think Team and figure 5.50(a) for CAD). Their corresponding controller has 485 states and 4774 transitions, and allows unlimited repetitions for the 5 scenarios, while the other 5 can only be executed a limited number of times.

The benefits of the cycle-based optimization become visible after 250000 executions, when the model learned for the Think Team component finally converges

to a precise-like model (see figure 5.49(a)). However, in the case of the CAD component (see figure 5.52(a)), the model still has only 5 identified cycles. The computed controller has 216 states and 1923 transitions.

Table 5.6: Model inference results – cycle-based exploration

nr.	TT size	TT tr.	CAD size	CAD tr.	ctrl size	ctrl tr.
500	120	543	170	1878	170	2080
	13	59	13	126		
1×10^3	66	295	196	2049	193	1580
	11	45	16	85		
1.5×10^3	40	141	222	2335	449	3596
	31	124	16	85		
2.5×10^3	49	190	242	2555	449	3596
	31	124	16	85		
5×10^3	49	193	254	2519	757	6000
	31	124	27	141		
1×10^4	27	88	250	2150	533	3488
	31	124	19	73		
2.5×10^4	22	57	249	2150	463	3297
	24	82	21	84		
5×10^4	21	53	245	1816	903	8879
	24	82	41	275		
1×10^5	20	52	193	1024	485	4774
	24	82	22	148		
2.5×10^5	11	36	196	1026	216	1923
	8	17	29	200		
5×10^5	11	36	205	1025	272	2746
	8	17	36	288		
7.5×10^5	11	36	133	1001	336	3682
	8	17	43	391		

After 750000 learning executions, the model of Think Team converges again, as expected, to the precise-like model. The model of the CAD component, which can be seen at figure 5.52(b), has a number of 5 identified cycles. Thus, the resulting controller, which has 336 states and 3682 transitions will allow for 5 out of 10 execution scenarios to repeat for an unlimited number of times. We notice here that, due to the high percentage of uncontrollable events in its alphabet,

the learning of the CAD component did no show any improvement while being subject to the cycle-based optimization.

The average trace length encountered during the normal exploration process was 18.46, with 2.46 more than 16, the sum of the two maximum trace lengths. It is worth mentioning here that the experiment using 1500000 executions has failed due to an Out of Memory exception thrown by the Java Virtual Machine.

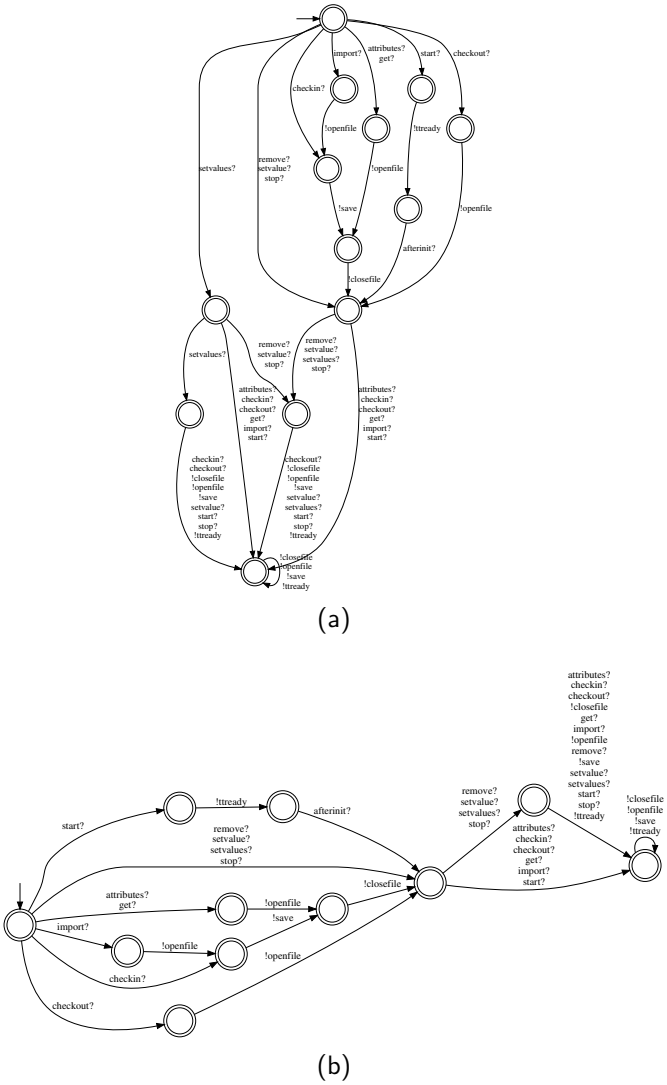
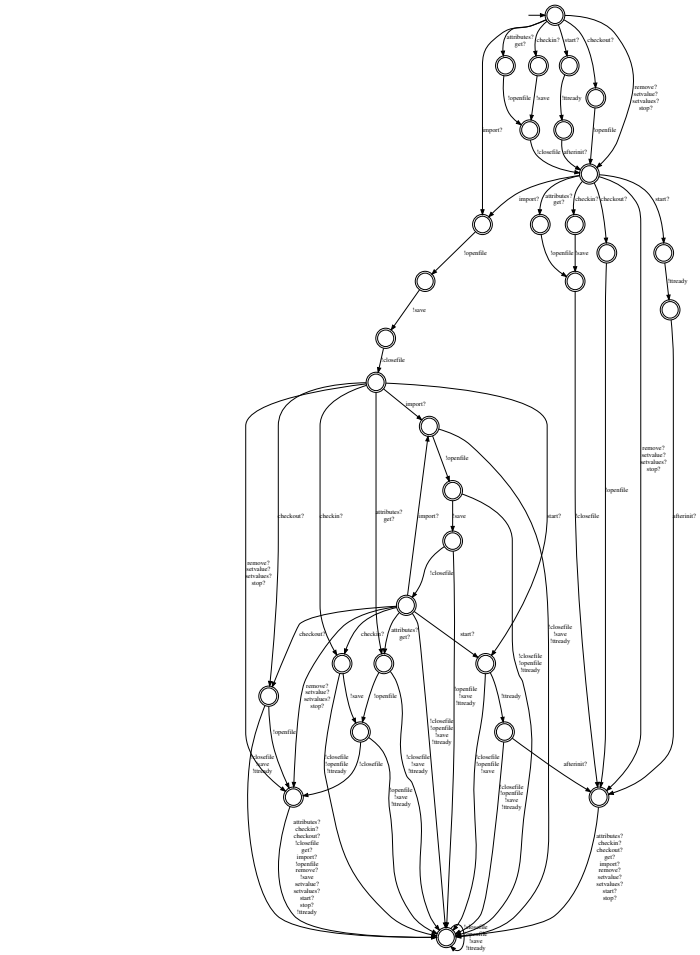
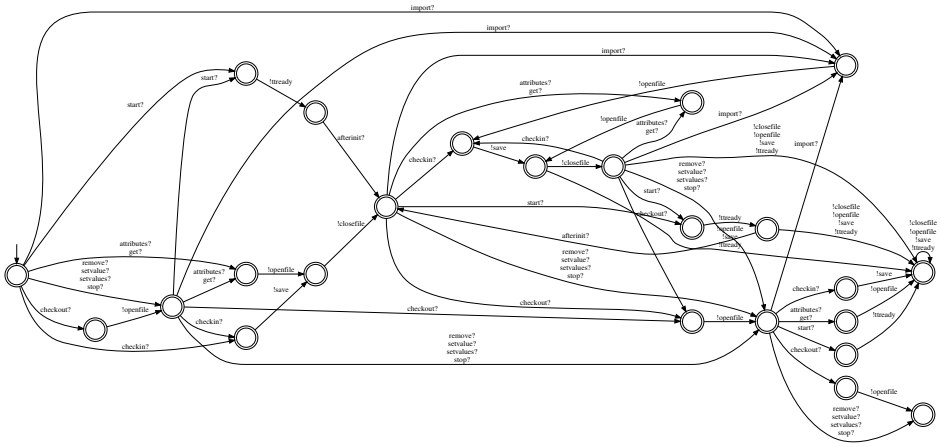


Figure 5.46: Learned models for component Think Team in: (a) 500 executions, (b) 1000 executions.

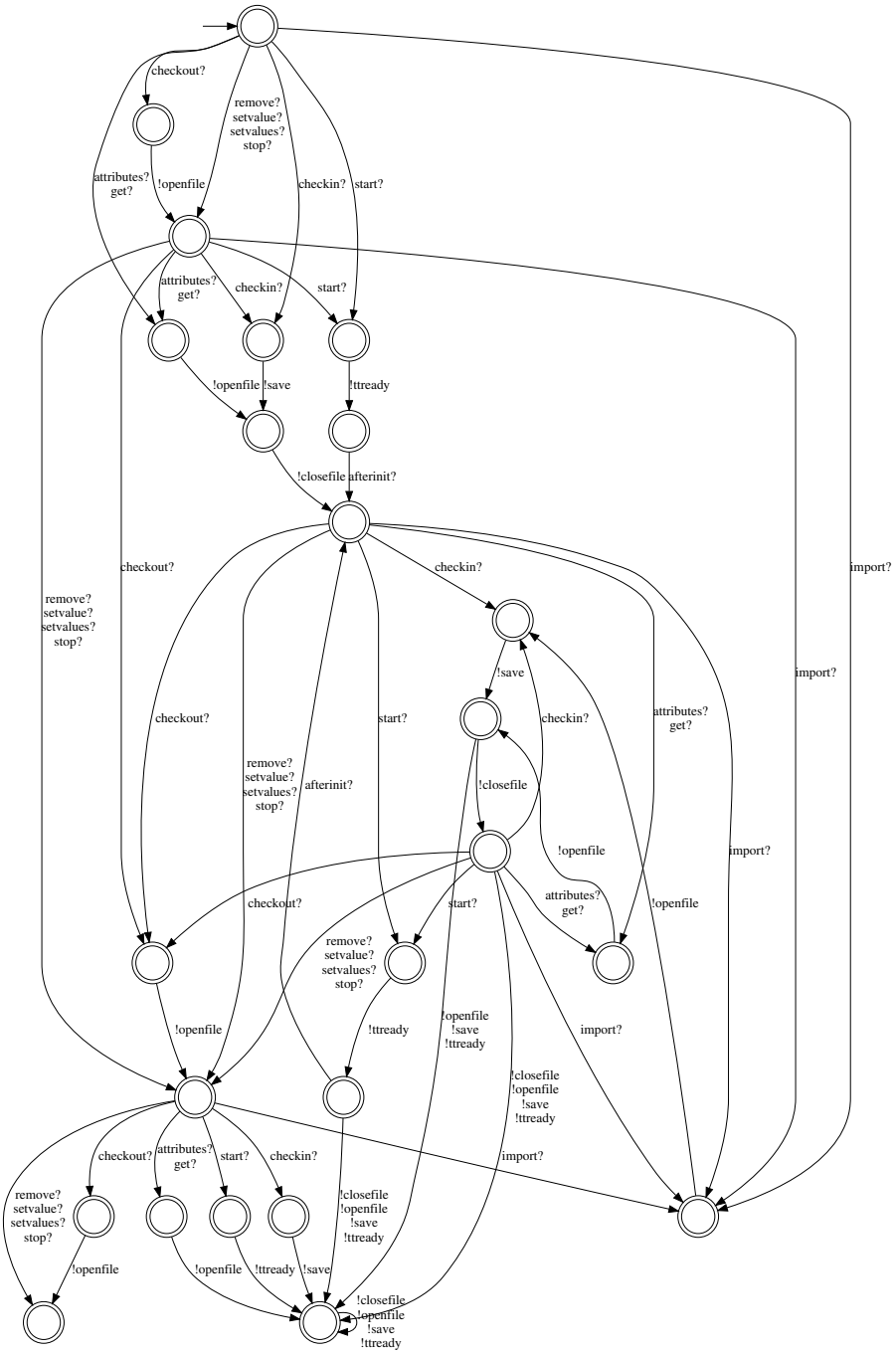


(a)



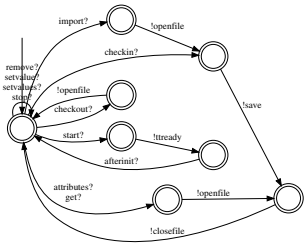
(b)

Figure 5.47: Learned models for component Think Team in: (a) 10 000 executions, (b) 25 000 executions.



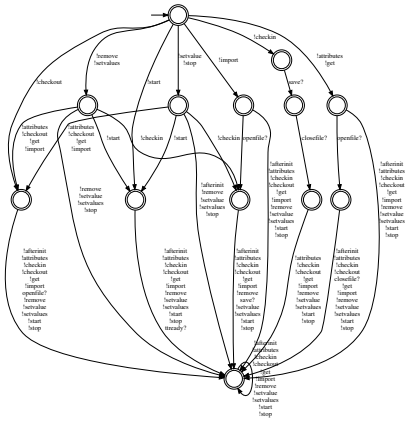
(a)

Figure 5.48: Learned model for component Think Team in: (a) 100 000 runs.

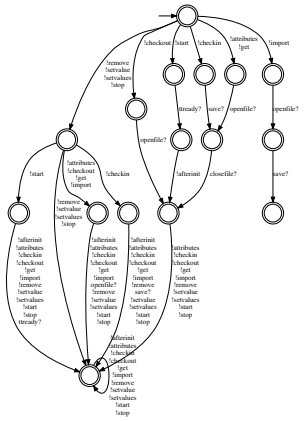


(a)

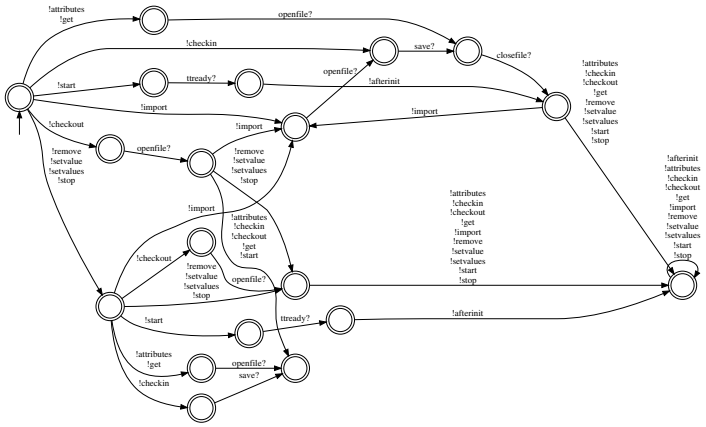
Figure 5.49: Learned model for component Think Team in: (a) 250 000 runs.



(a)

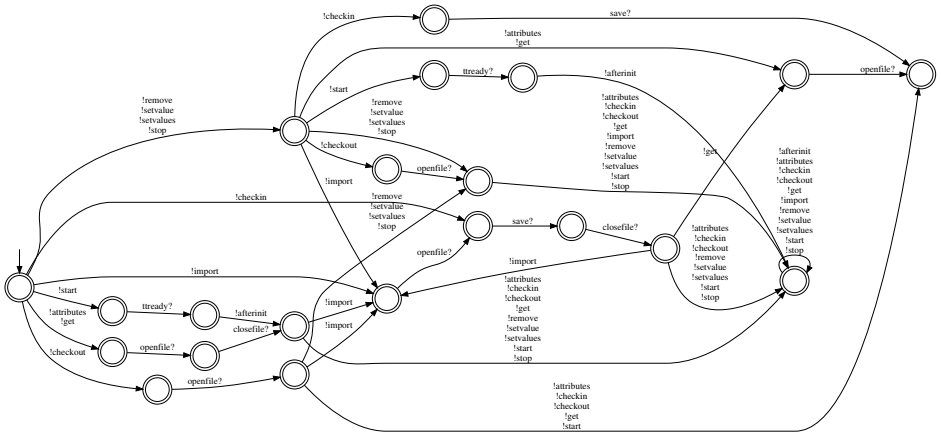


(b)

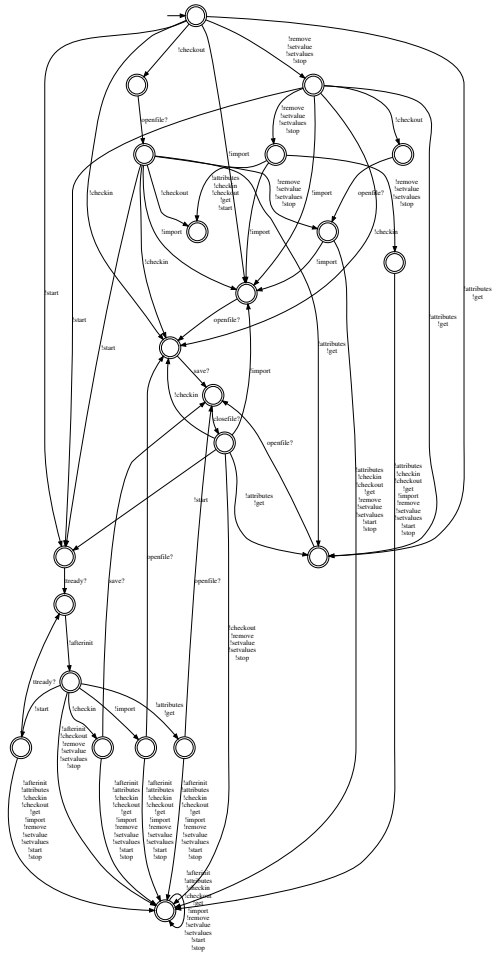


(c)

Figure 5.50: Learned models for component CAD in: (a) 500 executions, (b) 1000 executions, (c) 10 000 executions.

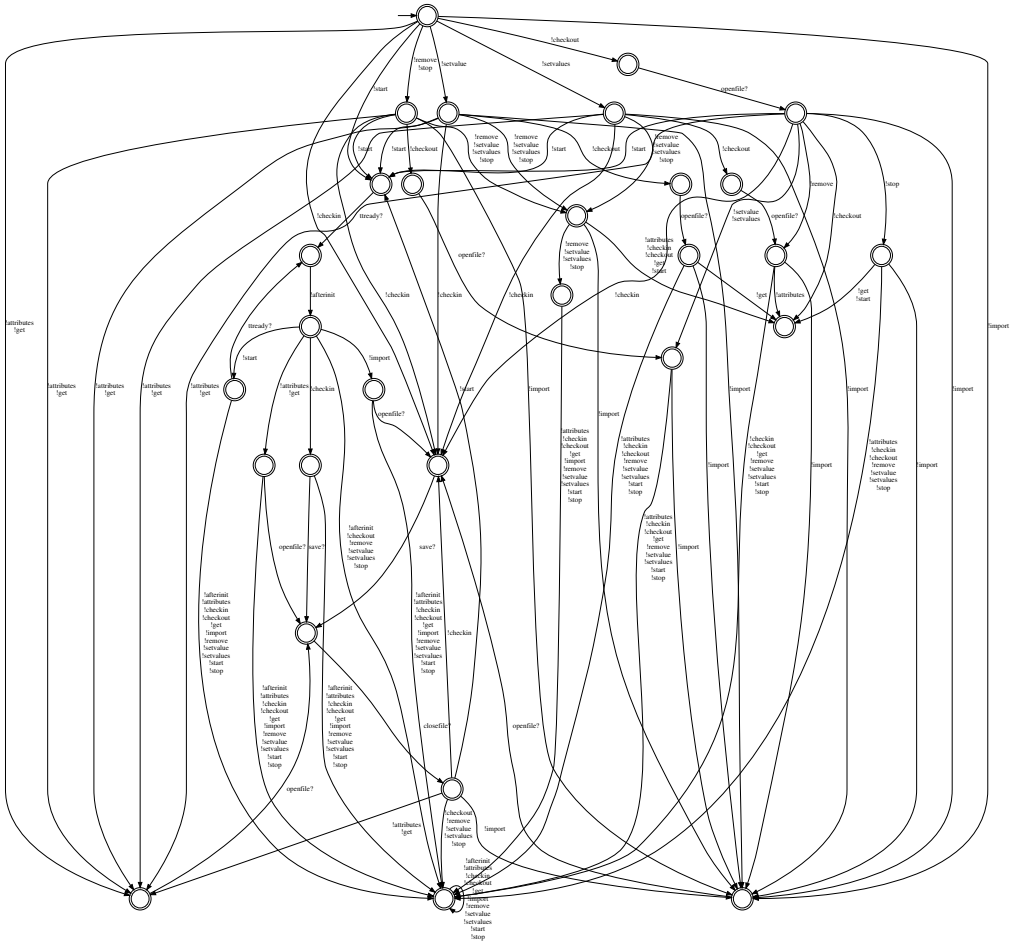


(a)

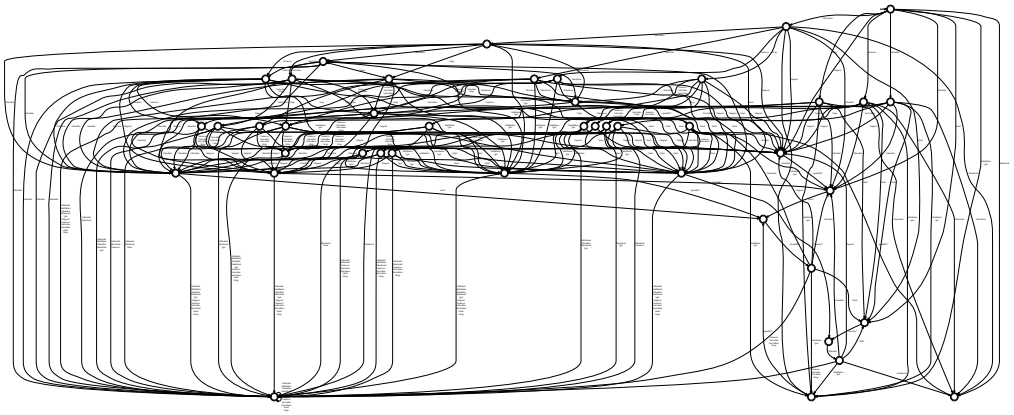


(b)

Figure 5.51: Learned models for component CAD in: (a) 25 000 executions, (b) 100 000 executions.



(a)



(b)

Figure 5.52: Learned models for component CAD in: (a) 250 000 executions, (b) 750 000 executions.

5.3.3 Distributed Exploration

As the Product Data Management System case study is all about a system with two components and a coordination policy, i.e. desired safety property, observations on the sets of performed experiments did not find significant differences between the models resulted from learning in a centralized versus a distributed exploration setting. The local exploration strategy still has, however, many relevant advantages: it is better adapted to the natural structure of distributed systems, it leads to shorter execution traces, and thus to a more efficient exploration than by the centralized strategy, and it also allows for a parallelization of the learning process.

Similarly to case of the centralized exploration setting, the experiments performed on this case study have used global size for the longest acyclic path $m = 4$. Also in this case, the fairness bounds were set independently for each component, thus $\theta = 12$ for the Think Team component and $\theta = 54$ for the CAD component. After each learning experiment, a controller has been computed for the inferred models and the coordination policy. We have then verified whether the obtained controller did work on the real system, i.e. on the plant determined by the precise models of participating components. For controller synthesis and verification we have used the Supremica tool [59].

Normal Exploration

The results for the unoptimized learning process are highly similar, as already mentioned, to the ones obtained in the centralized exploration experiment set. This is quite visible, for example, by comparing the summarized results from table 5.7, that presents the statistical data of models obtained by inference using the distributed exploration strategy, to the ones in table 5.5, that contains their correspondents obtained by centralized exploration. Not only the data regarding models obtained after the same number of executions is relatively similar, but some of the rows in these two tables are actually identical (the ones corresponding to 1000, 1500, 2500, 10000 and 100000 executions).

Some of the models learned in this set of experiments can be seen in figures 5.53 and 5.54 for the Think Team component, and in figures 5.55 and 5.56 for the CAD component. As their characteristics are highly similar to the specifics of the models already discussed in the centralized learning section, they won't be commented upon again here.

In the distributed exploration setting we have succeeded in reaching the upper bound on the number of allowed executions, 1500000, which ended up with a Out of Memory Exception in the centralized setting. As it can be seen in figure 5.54(d), the model learned for the Think Team component has finally converged to the precise-like automaton, while the model learned for CAD, visible in figure 5.56(d), although has not yet converged, has all its relevant cycles identified.

Table 5.7: Model inference results – normal exploration

nr.	TT size	TT tr.	CAD size	CAD tr.	ctrl size	ctrl tr.
500	176 17	772 71	166 13	1824 116	131	1524
1×10^3	255 14	1172 56	203 16	2126 85	129	1048
1.5×10^3	250 26	925 71	221 16	2324 85	129	1048
2.5×10^3	250 26	925 71	247 16	2610 85	129	1048
5×10^3	246 27	926 74	253 22	2545 127	177	1522
1×10^4	226 23	927 73	250 19	2150 73	324	1982
2.5×10^4	208 22	375 78	242 23	2149 106	346	2786
5×10^4	202 27	375 96	245 39	1774 264	625	6330
1×10^5	121 20	275 69	207 22	1024 148	419	4000
2.5×10^5	114 12	325 46	198 30	1025 210	224	2020
5×10^5	114 12	325 46	205 35	1025 269	264	2577
7.5×10^5	114 12	325 46	127 30	977 258	232	2421
1.5×10^6	72 8	401 17	114 16	932 109	120	1063

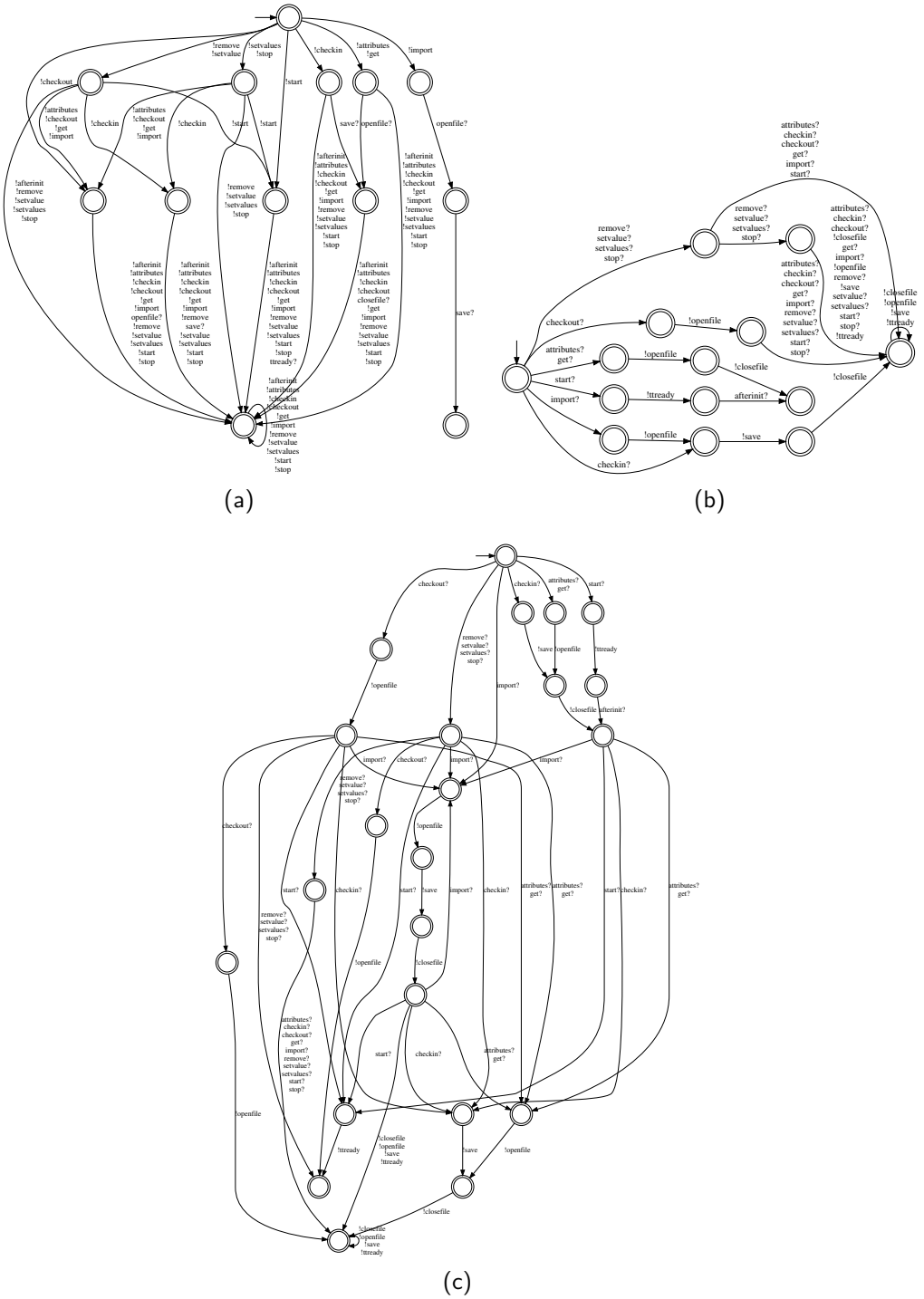


Figure 5.53: Learned models for component Think Team in: (a) 500 executions, (b) 1000 executions, (c) 10 000 executions.

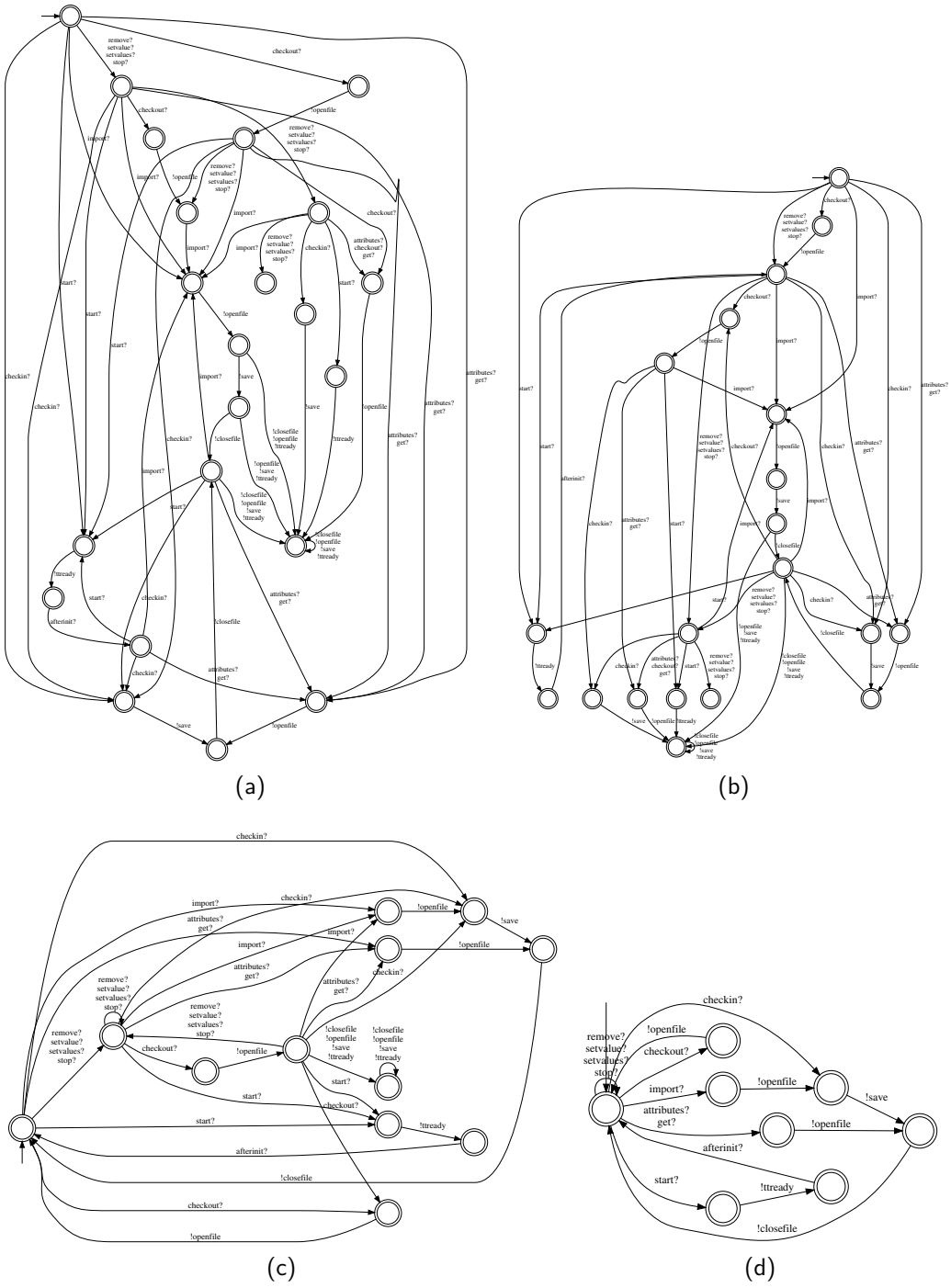


Figure 5.54: Learned models for component Think Team in: (a) 25 000 executions, (b) 100 000 executions, (c) 250 000 executions, (d) 1500 000 executions.

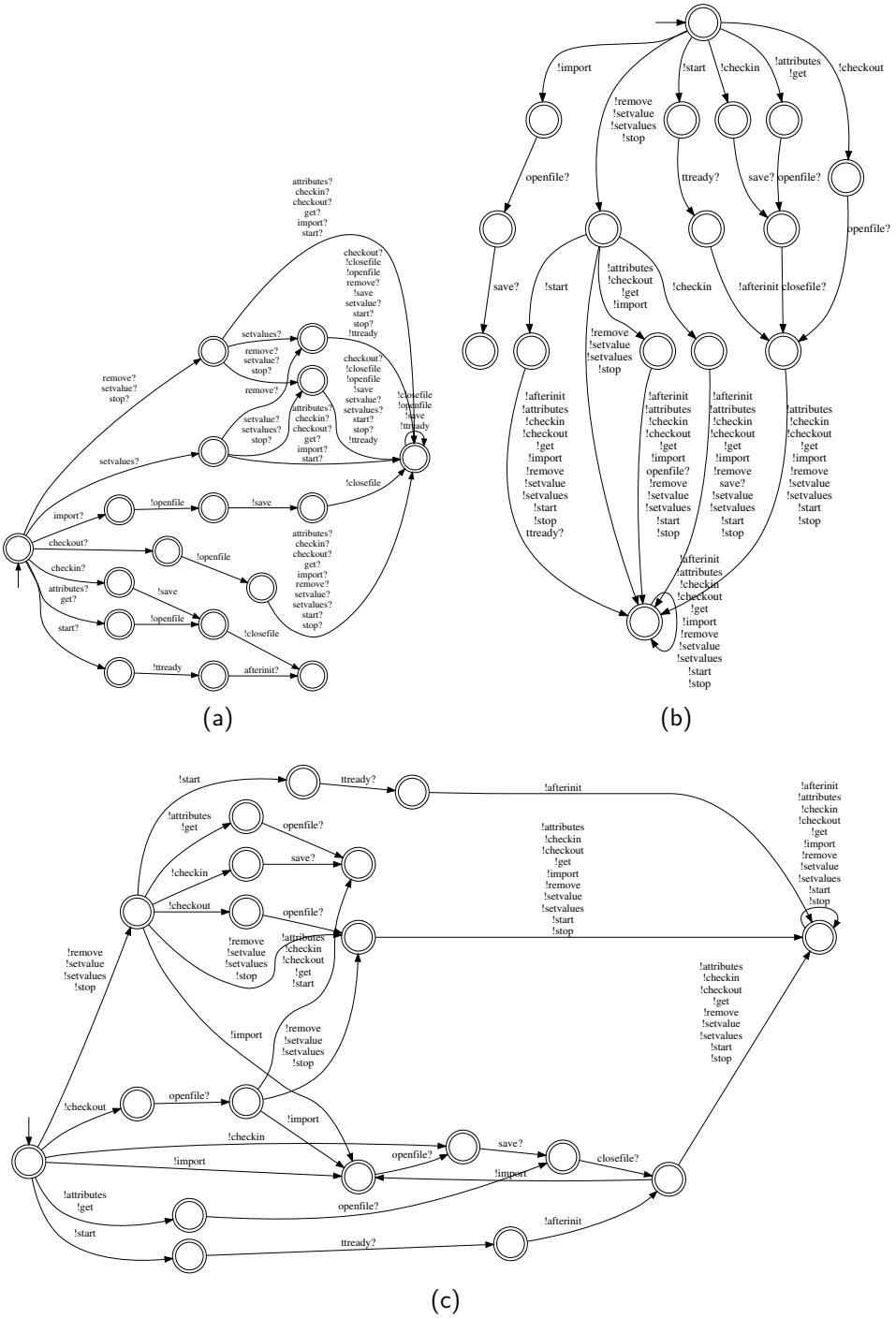


Figure 5.55: Learned models for component CAD in: (a) 500 executions, (b) 1000 executions, (c) 10 000 executions.

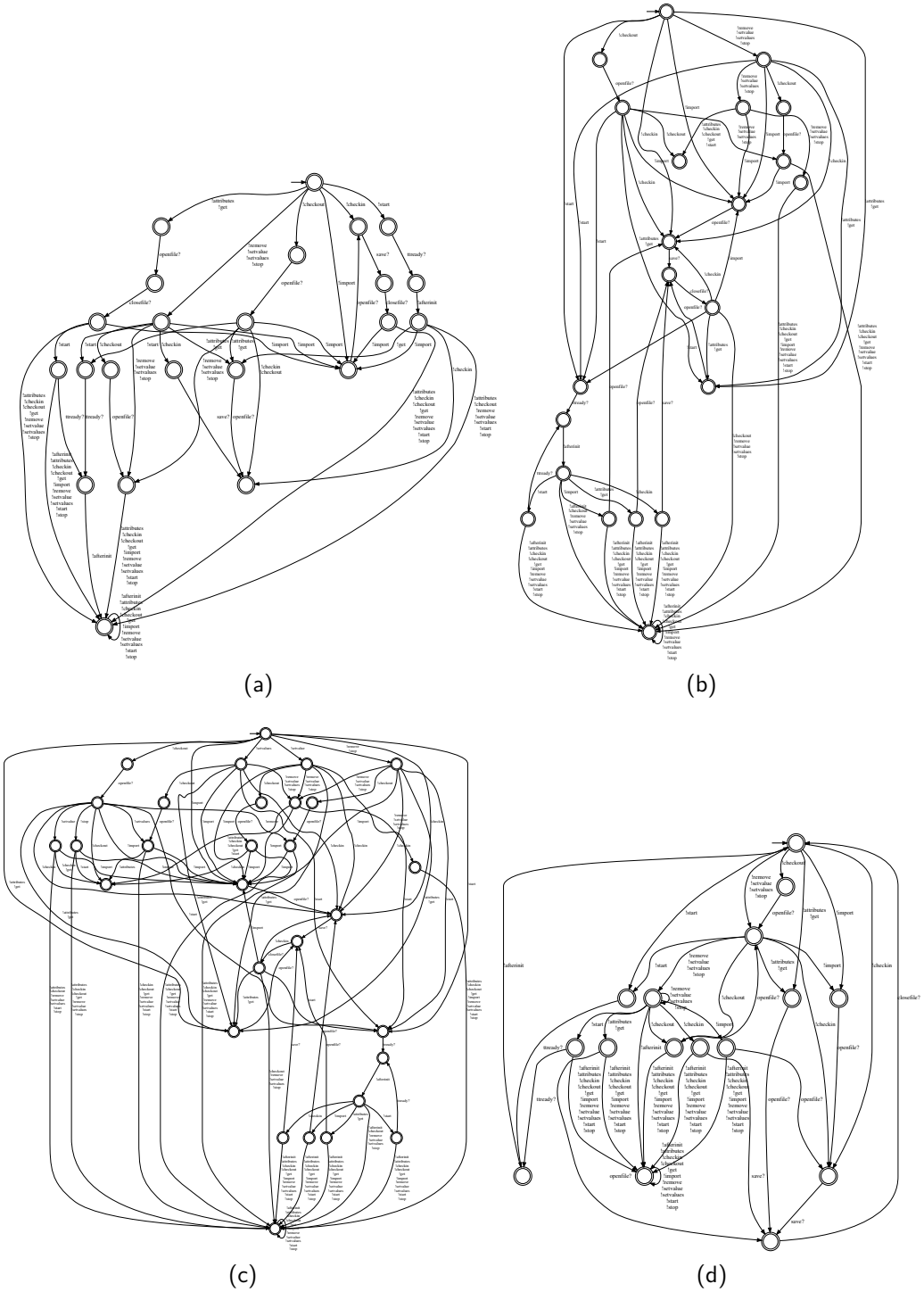


Figure 5.56: Learned models for component CAD in: (a) 25 000 executions, (b) 100 000 executions, (c) 250 000 executions, (d) 1500 000 executions.

Cycle-based Exploration

In the case of the cycle-based optimization of the learning process, the observed effects of the cycle-focused execution steering are relatively the same for the distributed learning setting as for the centralized one.

As expected, the models resulted are highly similar, to the ones obtained in the cycle-optimized version of the centralized exploration experiment set. This can be well observed, for example, by comparing the summarized results from table 5.8, that presents the statistical data of models obtained by a learning process using the distributed exploration strategy, to the ones in table 5.6, that contains their correspondents learned by centralized exploration. Not only the data regarding models obtained after the same number of executions is relatively similar, but some of the rows in these two tables are actually identical (the ones corresponding to 1000, 1500, 10000 and 100000 executions).

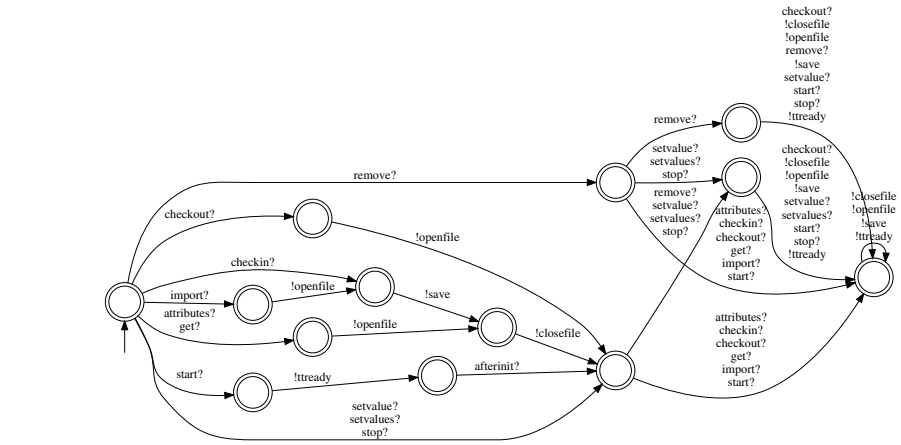
Some of the models learned in this set of experiments can be seen in figures 5.57 and 5.58 for the Think Team component, and in figures 5.59 and 5.60 for the CAD component. As their characteristics are highly similar to the specifics of the models already discussed in the centralized learning section, they won't be commented upon again here.

We should however emphasize that the model of the Think Team component does converge to a precise-like model after a number of 250000 executions in the cycle-optimized version, instead of taking between 750000 and 1500000 executions to reach convergence, as it is the case with the unoptimized learning algorithm in the distributed setting.

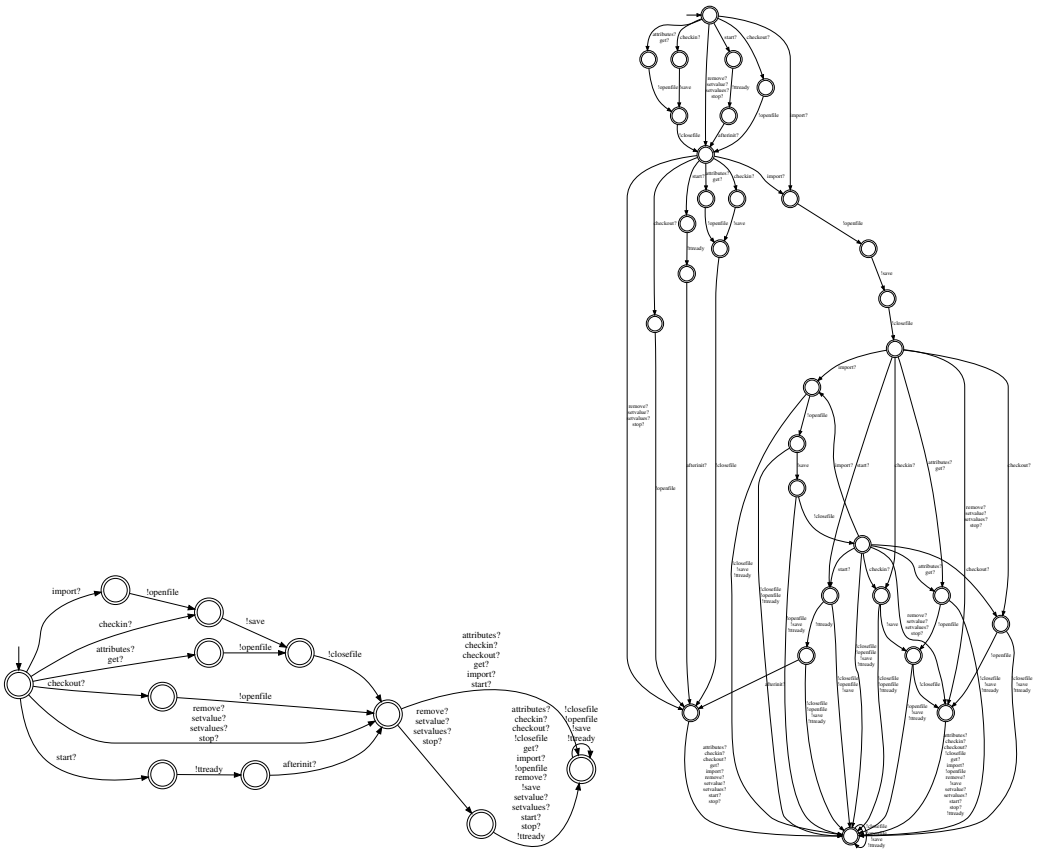
Also, in the experiments using the distributed exploration setting, we have succeeded in reaching the upper bound on the number of allowed executions, 1500000, which ended up with a Out of Memory Exception in the centralized setting. As it can be seen in figure 5.60(d), the model learned for CAD, although has not yet reached convergence to a precise-like automaton, has all its relevant cycles identified, 5 of which being found from its actual initial state. Since the model learned for the Think Team component has already converged to a precise-like model since the threshold of 250000 executions has been reached, the resulting controller for the models learned in 1500000 executions will allows for all relevant execution scenarios to occur for an unlimited number of times. This almost-ideal computed controller is not only permissive on the real system, but also relatively small in size, compared to some of the previously obtained controllers, as it only has 120 states and 1063 transitions (see table 5.8).

Table 5.8: Model inference results – cycle-based exploration

nr.	TT size	TT tr.	CAD size	CAD tr.	ctrl size	ctrl tr.
500	120 13	543 59	162 17	1745 129	222	2272
1×10^3	123 11	569 45	188 16	1961 85	193	1580
1.5×10^3	30 29	92 113	220 16	2313 85	449	3596
2.5×10^3	28 28	89 114	245 16	2588 85	417	3346
5×10^3	39 31	148 124	253 24	2548 129	673	5436
1×10^4	52 31	219 124	250 19	2150 73	533	3488
2.5×10^4	24 24	74 82	239 22	2148 96	485	3630
5×10^4	20 24	52 82	244 35	1855 233	771	7541
1×10^5	24 24	74 82	208 22	1024 148	485	4774
2.5×10^5	11 8	36 17	195 28	1027 190	208	1826
5×10^5	11 8	36 17	205 35	1024 269	264	2577
7.5×10^5	11 8	36 17	132 33	1002 287	256	2712
1.5×10^6	9 8	27 17	114 16	932 109	120	1063



(a)



(b) (c)

Figure 5.57: Learned models for component Think Team in: (a) 500 executions, (b) 1000 executions, (c) 10 000 executions.

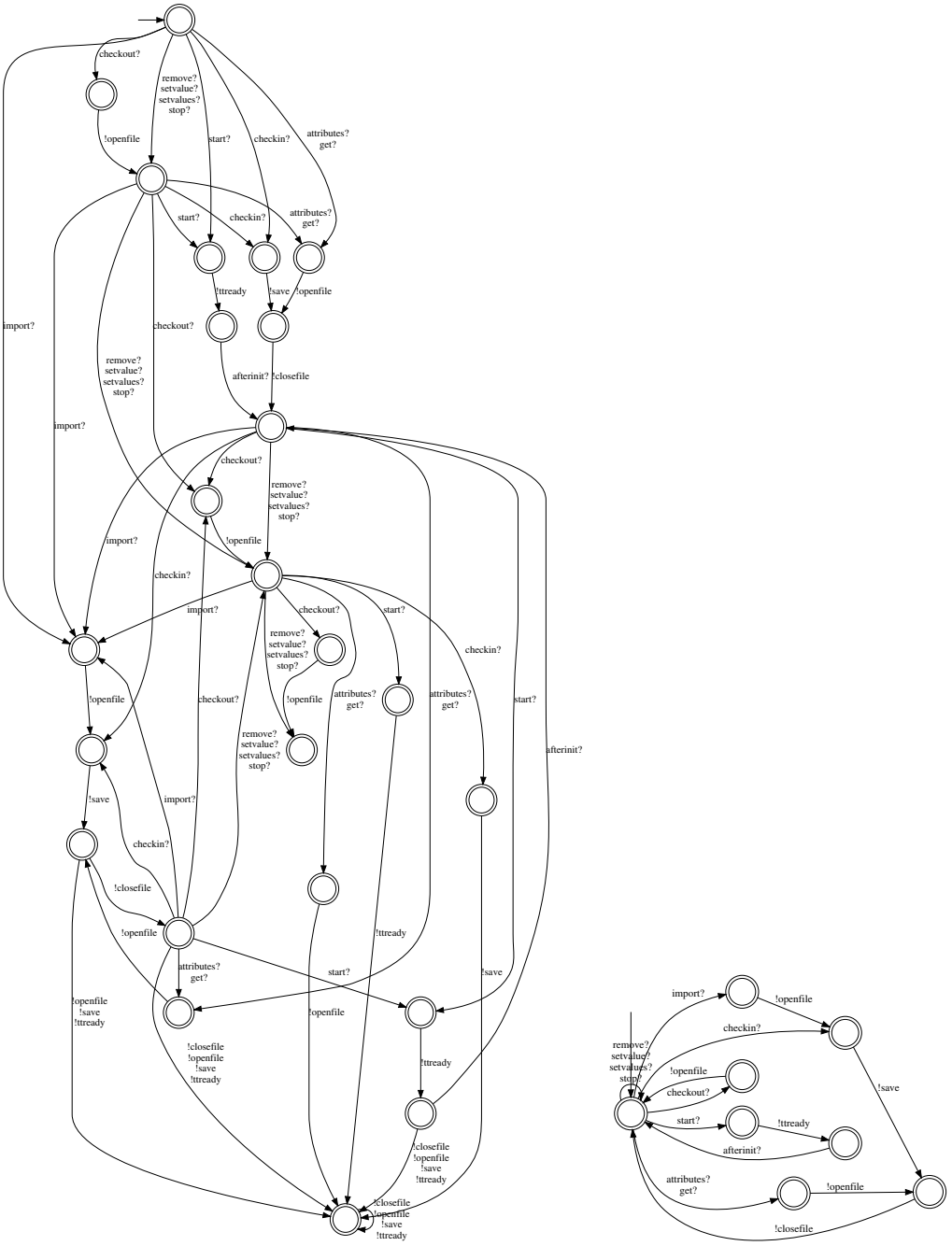


Figure 5.58: Learned models for component Think Team in: (a) 25 000 executions, (b) 250 000 executions.

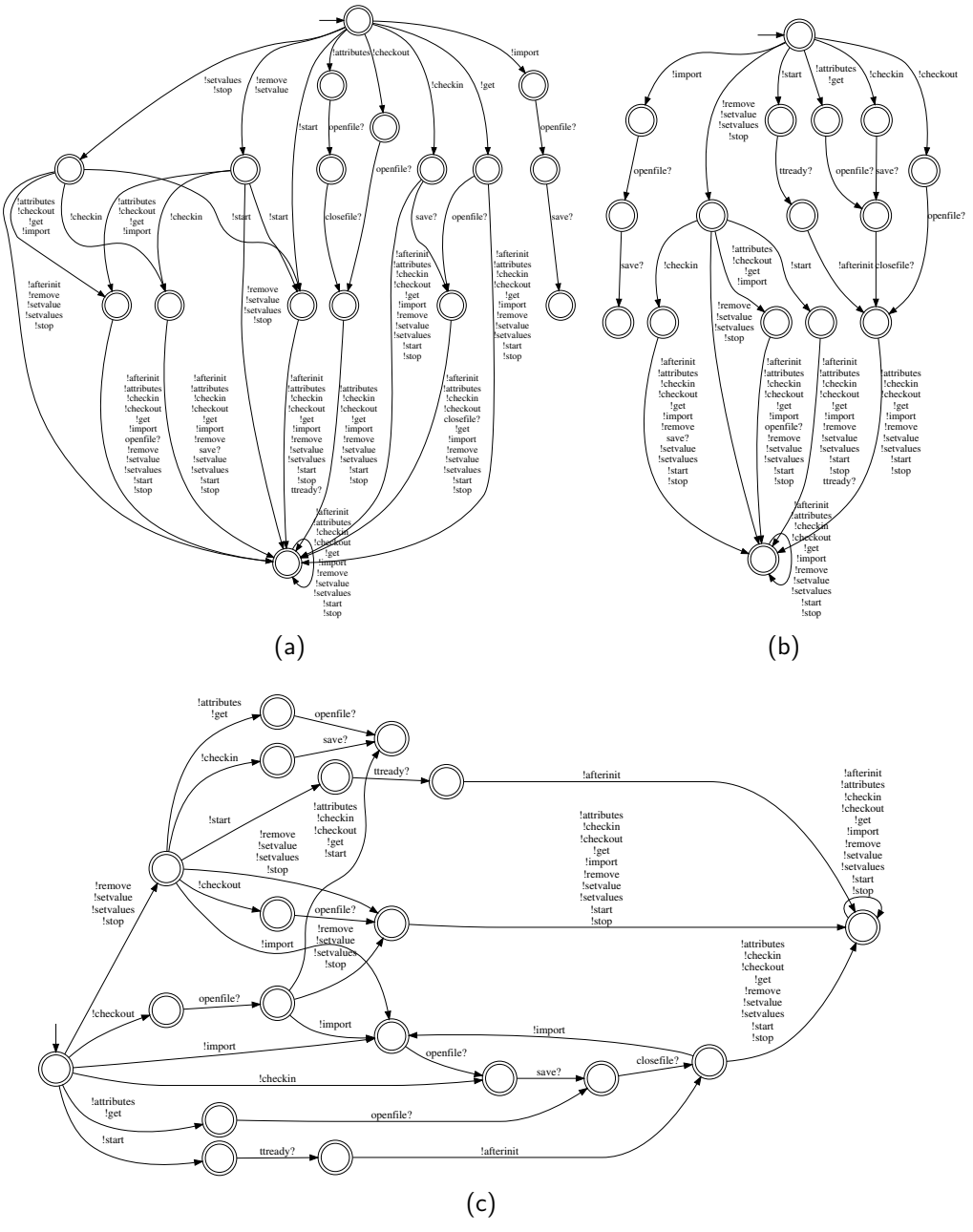


Figure 5.59: Learned models for component CAD in: (a) 500 executions, (b) 1000 executions, (c) 10 000 executions.

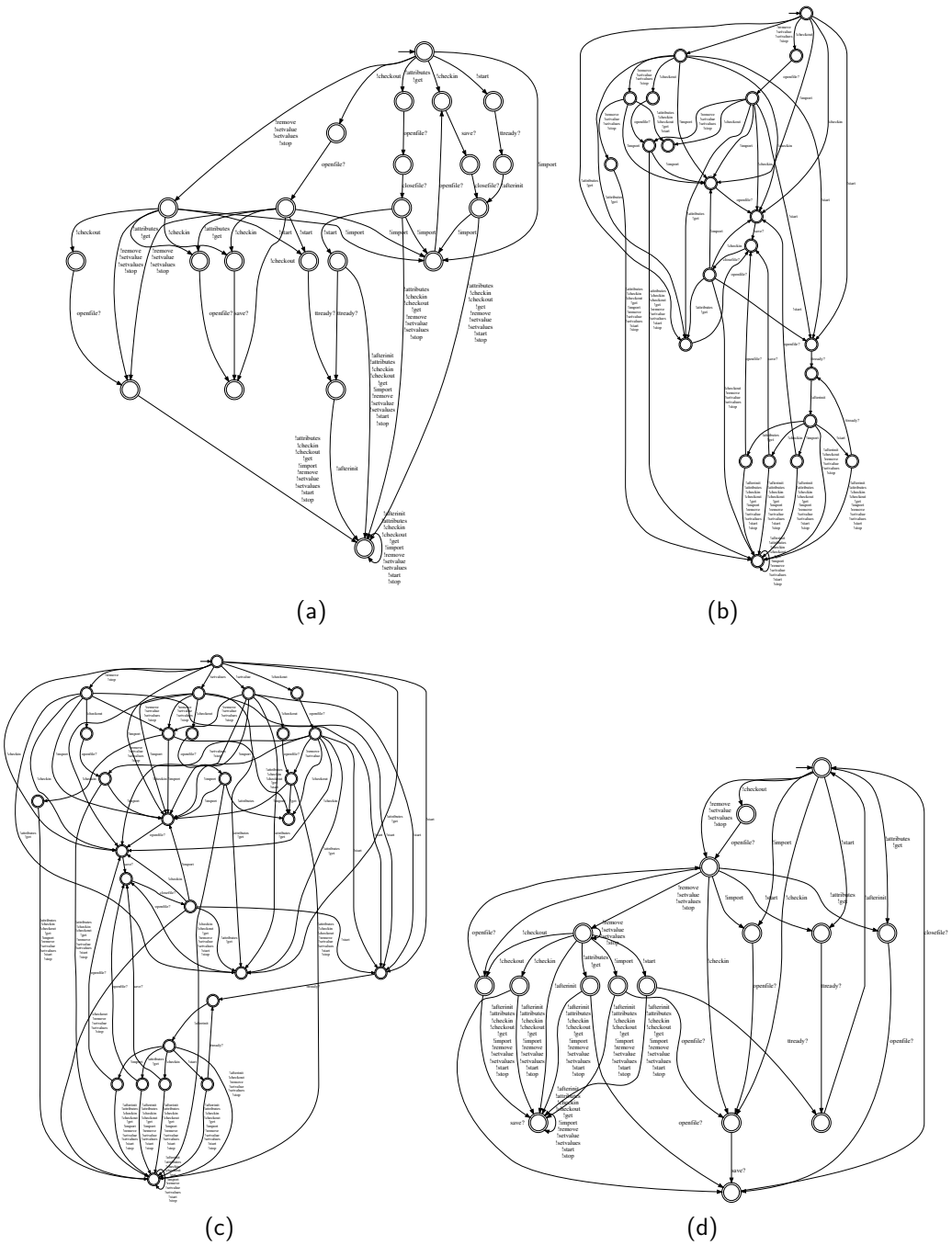


Figure 5.60: Learned models for component CAD in: (a) 25 000 executions, (b) 100 000 executions, (c) 250 000 executions, (d) 1500 000 executions.

5.4 Case Study: Domotics

Domotics is a system for home automation, that aims to provide smart home services, such as light, climate or multimedia device control. This case study is part of a much larger project ArchiteCture for Smart Environment (ACSE) at Orange Labs, that contains several prototypes for building automation.

The Domotics Case Study has been experimented upon by Shahbaz in [74], which inferred Mealy machine models for the participating devices. We rewrote two of the device models as automata with send and receive events, and built an asynchronous abstract version of these devices.

The devices considered for this case study are a Media Renderer, that mainly deals with rendering Audio/Video content from the home network, and a Light Control System, which controls the illumination of the building, by reducing or increasing it at request.

The alphabet of the Media Renderer component is as follows:

- pon?: receiving this message turnes the Media Renderer on
- ponok!: sent when the Media Renderer has been successfully initialized
- poff?: receiving this message turnes the Media Renderer off
- poffok!: sent when the Media Renderer has been successfully turned off
- play?: requests the MR to play a specific Audio/Video file
- playok!: the *play* request has been acknowledged and is going to be fulfilled
- pause?: requests the MR to pause a specific Audio/Video file that is currently playing
- pauseok!: the *pause* request has been acknowledged and is going to be fulfilled
- stop?: requests the MR to stop a specific Audio/Video file that is currently playing
- stopok!: the *stop* request has been acknowledged and is going to be fulfilled

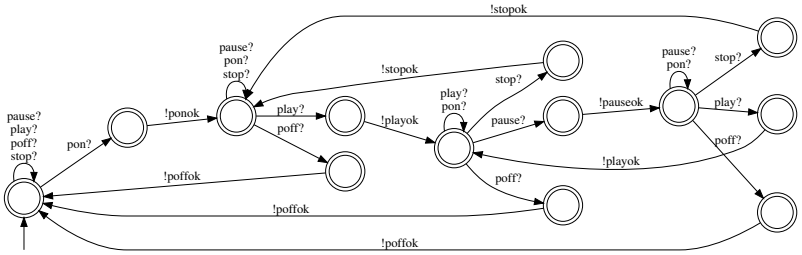
The alphabet of the Light Control System component is composed of the following events:

- lon? : receiving the *lon* message turns the Light Control System on
- lonok!: message sent when the LCS has been successfully turned on
- loff?: turns the LCS off
- loffok!: message sent when the LCS has been successfully turned off
- dim?: commands reducing light intensity with one level
- bright?: commands increasing light intensity with one level
- dim1!: sent when light has been dimmed to level 1
- dim2!: sent when light has been dimmed to level 2
- dim3!: sent when light has been dimmed to level 3
- bright2!: sent when light intensity has been increased to level 2
- bright3!: sent when light intensity has been increased to level 3
- bright4!: sent when light intensity has been increased to level 4

In the original case study, the messages sent to and received from the two main Domotics devices, the Media Renderer and the Light Control System, are handled by a TV_ Ctrl device. However, since the Media Renderer and the Light Control System do not share any common events, the TV_ Ctrl behaviour is most general and no desired property is defined for the global system, we have decided to split the responsibilities of the TV_ Ctrl device into two similar devices: TCMR, that communicates to the Media Renderer, and TCLS, that communicates to the Light Control System.

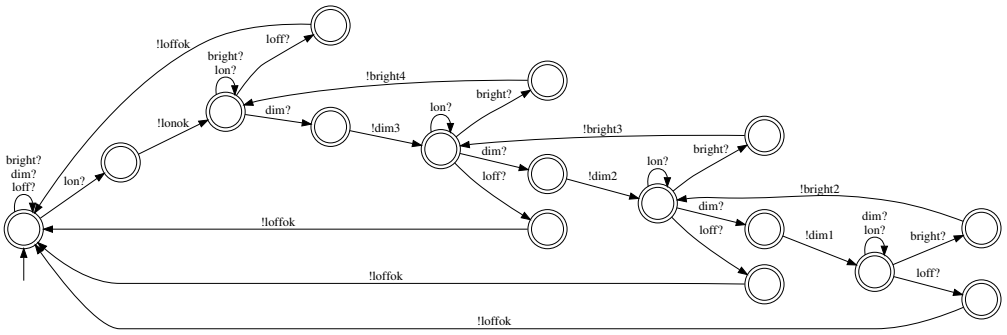
Thus, we have basically two separate subsystems, the Media Renderer and its remote control, respectively the Light Control System and its remote control. Further on, in our experiment, the MR and LCS components are regarded as black boxes and their models are learned by BASYL. Since no temporal property has been specified for any of the resulting systems, we consider it to be as most general for both cases. Therefore, no controllers are needed in this situation, the only thing we are interested in are the inferred models.

The figure 5.61(a) shows us the original model of component Media Renderer. Similarly, the original model of the Light Control System can be seen in figure 5.62(a). The Media Renderer component has an alphabet of 10 events, out of which 5 are controllable and 5 are uncontrollable, while the Light Control System component has an alphabet of 12 events, from which only 4 events are



(a)

Figure 5.61: (a) MR original model



(a)

Figure 5.62: (a) LCS original model

controllable and can be enabled at will by sending an expected message, while the remaining 8 events are uncontrollable.

The relevant parameters for the two black box components considered were set as following: m, the upper bound on the length of acyclic traces, is set to $m = 8$ for the Media Renderer component and to $m = 10$ for the Light Control System component, and θ, the fairness bound, becomes $\theta = 18$ for the Media Renderer, and $\theta = 34$ for the Light Control System.

The experiments performed for the Domotics case study use both the regular learning algorithm for incomplete model learning and its optimized, cycle-based version. The complete learning algorithm has not been experimented with, due to the relatively large values of the acyclic trace bound m and high fairness values θ of both the Media Renderer and the Light Control System black boxes. These relatively high values of the relevant parameters would have seriously affected the feasibility of complete learning experiments. For cycle-based exploration, as

no desired property was specified, the important cyclic scenarios for each of the two black boxes were manually derived from their precise models.

The specification of the cycles considered for component Media Renderer is presented below. The cycles considered have three join points, from which the execution can pass from one cycle to the other. The first scenario describes the turning on and off of the Media Renderer, the second says that the component can be turned on, made to play a current media file, and turned off, while the third describes the component being turned on, made to play a file, then paused and turned off. All these cycles are expected to occur from the initial state – the first join point. The second join point appears after the access trace represented by the component being turned on. From there, for an unlimited number of times, a media file can be played and stopped, or played, paused and stopped. Finally, the third join point is situated after an access trace represented by the Media Renderer being turned on and made to play a media file, after which the file can be paused and stopped for an unlimited number of times.

```
:[1]<pon?><ponok!>[2]<poff?><poffok!>

:[1]<pon?><ponok!>[2]<play?><playok!>[3]<poff?><poffok!>

:[1]<pon?><ponok!>[2]<play?><playok!>[3]<pause?><pauseok!>
<poff?><poffok!>

<pon?><ponok!>:[2]<play?><playok!>[3]<stop?><stopok!>

<pon?><ponok!>:[2]<play?><playok!>[3]<pause?><pauseok!>
<stop?><stopok!>

<pon?><ponok!><play?><playok!>:[3]<pause?><pauseok!>
<stop?><stopok!>
```

Below, the specified cycles for the Light Control System component are presented. The cycles considered for the more complex component of the Domotics case study have 4 join points. From the initial state – the first join point, the lights can be switched on and off, or can be switched on, dimmed once and switched off, dimmed twice and switched off, or dimmed three times and then switched off. The second join point arises after the lights are switched on: they can be dimmed and then brightened back for an unlimited number of times. The third join point has an access trace represented by turning the Light Control System on and dimming the lights – the lights can then be dimmed again

and brightened back for an unlimited number of times. And finally, the last join point appears after an access traces describing the lights being switched on and then dimmed twice – they can then be dimmed again and then brightened back in the last considered loop of the cycle specification file.

```
:[1]<lon?><lonok!>[2]<loff?><loffok!>

<lon?><lonok!>:[2]<dim?><dim3!>[3]<bright?><bright4!>

<lon?><lonok!><dim?><dim3!>:[3]<dim?><dim2!>
[4]<bright?><bright3!>

<lon?><lonok!><dim?><dim3!><dim?><dim2!>:[4]<dim?><dim1!>
<bright?><bright2!>

:[1]<lon?><lonok!>[2]<dim?><dim3!>[3]<loff?><loffok!>

:[1]<lon?><lonok!>[2]<dim?><dim3!>[3]<dim?><dim2!>[4]<loff?>
<loffok!>

:[1]<lon?><lonok!>[2]<dim?><dim3!>[3]<dim?><dim2!>[4]<dim?>
<dim1!><loff?><loffok!>
```

Media Renderer – Normal Exploration

The summarized results of incomplete model inference for the Media Renderer component, obtained using the normal exploration strategy, can be seen in table 5.9, in both unminimized and minimized form. The models obtained were too large to be properly visualized.

From the original Media Renderer model (see figure 5.61(a)), we can see that all 5 controllable events in the component's alphabet can be enabled from any state. This leads to a very dense execution trace tree, which is also hard to partition and to build a model from, due to the relatively high values of the relevant parameters: $m = 8$, and $\theta = 18$. By comparison, the component SSO Client from the Single Sign On case study has lower, but somehow comparable parameter values, at $m = 6$ and $\theta = 12$, however its execution trace tree is remarkably sparse, which makes it easy to learn. But the resulting dense trace tree obtained when learning the Media Renderer component brings the time complexity of equivalence partitioning and model building close to its upper

bound of $|\Sigma|^{3m}$, which for $|\Sigma| = 10$ and $m = 8$ become rather difficult to handle. To avoid having the equivalence partitioning phase take too much computing time, the number of learning traces has been severely restricted, so that the trace tree will be only partially filled.

Thus, as it can been seen in table 5.9, the number of learning executions used for the Media Renderer cases study has been limited to 5000, which hardly allows a complete exploration for vertices at depths larger than 3. This doesn't leave much room for cycle identification, however, in this case, the model learning algorithm still allows us to discover some safe, usable paths in the behaviour of the Media Renderer black box component.

Table 5.9: Model inference results – normal exploration

nr.	MR size	MR tr.
200	867 255	5027 1640
250	1020 283	5906 2089
500	1842 310	10591 2089
1×10^3	3056 477	17671 3407
1.5×10^3	6015 560	34362 3953
2×10^3	4033 528	23405 3962
2.5×10^3	5010 561	29268 4356
5×10^3	9155 646	52976 4917

Media Renderer – Cycle-Based Exploration

The cycle-based exploration offers an interesting improvement over the unoptimized exploration, as it focuses the relatively reduced number of available executions to several interesting usage scenarios. As a result, the obtained models are significantly smaller than the ones learned by the normal exploration strategy,

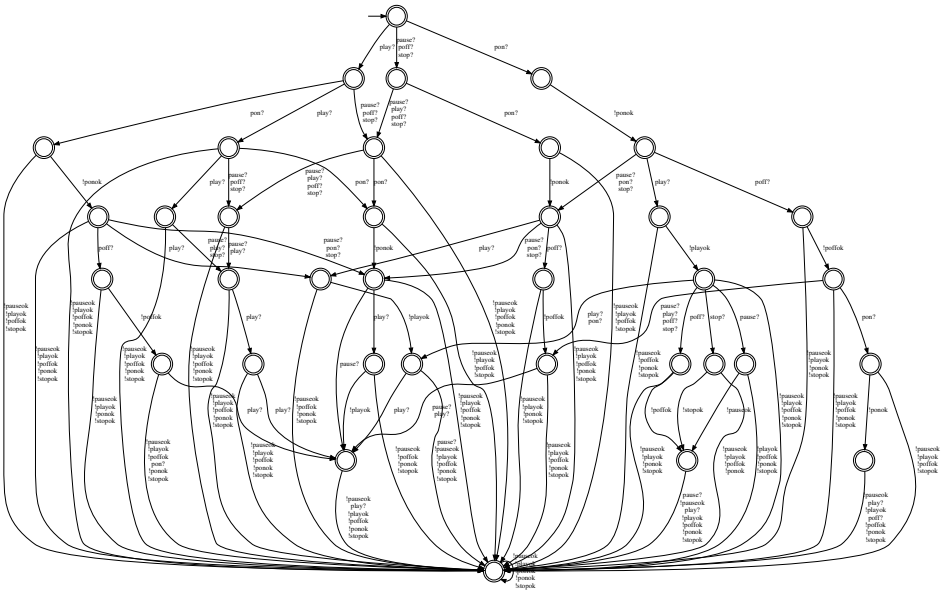
as it can be observed from the statistical data presented in table 5.10, and by comparing these model sizes to the ones in table 5.9.

The specification of interesting cycles is given again below.

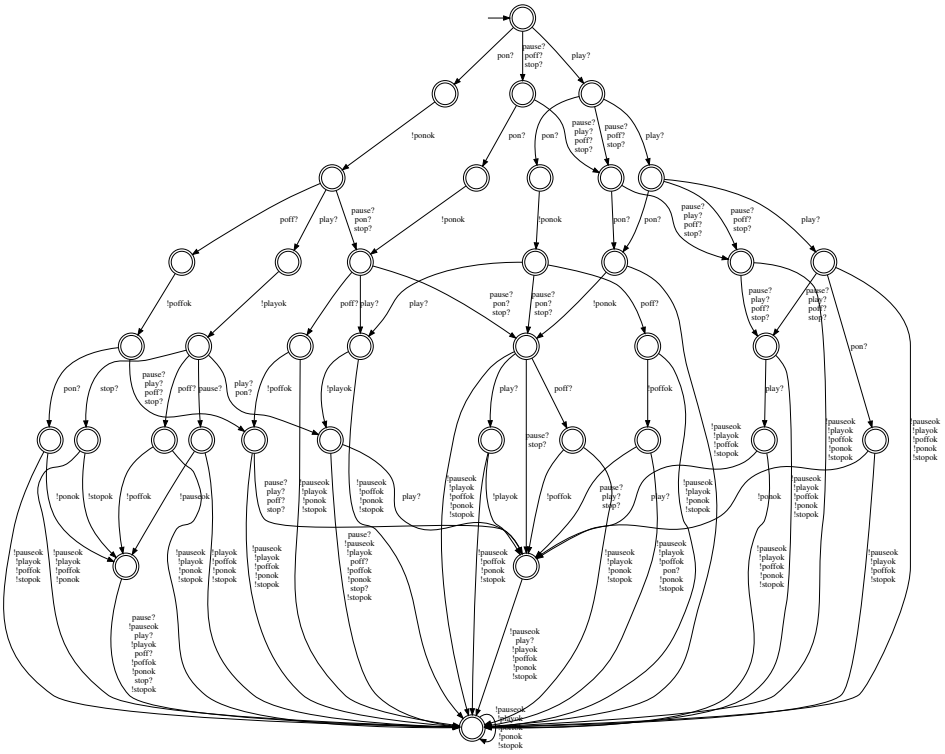
```
:[1]<pon?><ponok!>[2]<poff?><poffok!>

:[1]<pon?><ponok!>[2]<play?><playok!>[3]<poff?><poffok!>

:[1]<pon?><ponok!>[2]<play?><playok!>[3]<pause?><pauseok!>
<poff?><poffok!>

<pon?><ponok!>:[2]<play?><playok!>[3]<stop?><stopok!>

<pon?><ponok!>:[2]<play?><playok!>[3]<pause?><pauseok!>
<stop?><stopok!>

<pon?><ponok!><play?><playok!>:[3]<pause?><pauseok!>
<stop?><stopok!>
```

Table 5.10: Model inference results – cycle-based exploration

nr.	MR size	MR tr.
200	505 37	2939 228
250	592 36	3447 226
500	380 37	2157 197
1×10^3	593 36	3408 203
1.5×10^3	755 38	4380 221
2×10^3	917 37	5352 218
2.5×10^3	244 39	1332 166
5×10^3	343 41	1919 183



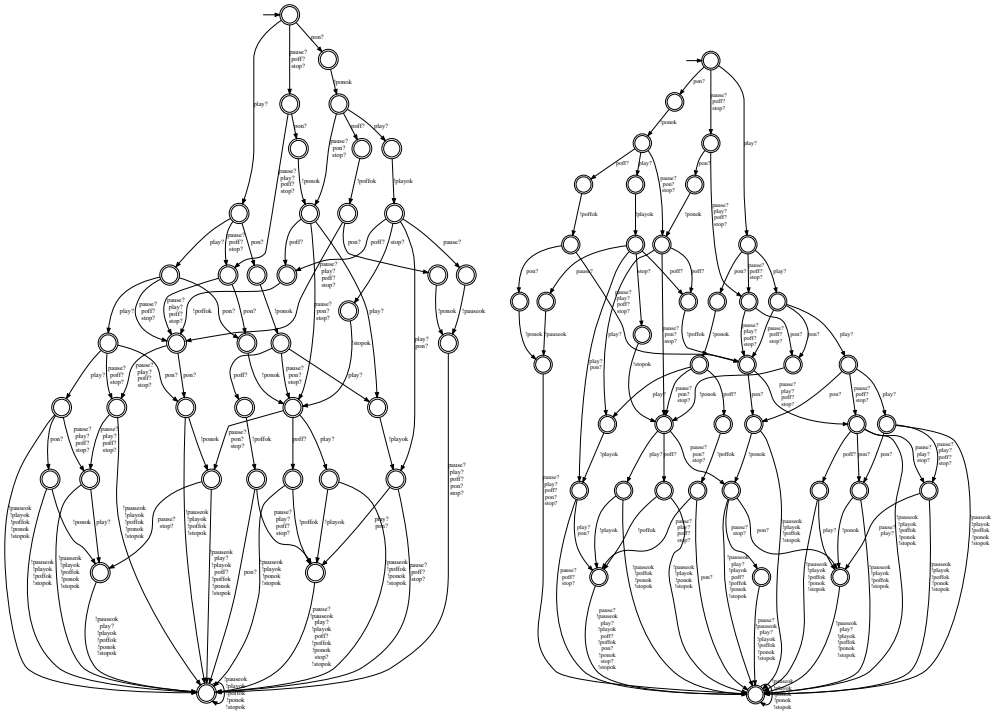
(a)



(b)

Figure 5.63: Learned models for component Media Renderer in: (a) 250 executions, (b) 500 executions.

Some of the models obtained during the cycle-optimized learning of the Media Renderer component are also presented. In figure 5.63(a) we can see the model of the Media Renderer obatined in 250 learning executions, while figure 5.63(b) present a model learned in 500 executions. Further on, in figure 5.64(a) we can see a model of Media Renderer Learned in 2500 executions, while the model in figure 5.64(b) has been learned in 5000 executions. Although only the relevant scenarios described in the cycle specified have been explored, none of these cycles has been identified within the maximum 5000 executions available. However, the models obtained by focusing on the relevant scenarios are not only smaller than their unoptimized correspondents, but also quite well explored for the low number of executions employed. Thus, the model obtained after 250 executions has well-explored states up to a depth of 4, the one learned in 500 executions has a significant number of well-explored states also at depth 5, while the number of outgoing uncontrollable transitions to the unknown-future state decreases further more after 2500 and 5000 learning executions. The latter models have completely explored states at depths up to 6 and 7.



(a) (b)

Figure 5.64: Learned models for component Media Renderer in: (a) 2500 executions, (b) 5000 executions.

Light Control System – Normal Exploration

The summarized results of incomplete model inference for the Light Control System component, obtained using the normal exploration strategy, can be seen in the below table 5.11, in both unminimized and minimized form. Like in the case of the Media Renderer component, the models obtained were too large to be properly visualized.

Similarly to the Media Renderer case study, by studying the original Light Control System model (see figure 5.62(a)), we can see that all 4 controllable events in the component's alphabet can be enabled from any state. Further on, what distinguishes the Light Control System component from the Media Renderer is that it has a larger number of uncontrollable events, 8, and only 4 controllable events, which would make it harder to control properly if it wouldn't already be an output deterministic component. It also has a significantly larger fairness bound, $\theta = 34$, and a larger bound on the longest acyclic path set at $m = 10$.

Table 5.11: Model inference results – normal exploration

nr.	LCS size	LCS tr.
200	1135 367	9920 3350
250	1386 421	12111 3865
500	2315 604	20286 5691
1 $\times 10^3$	4163 660	36315 6491
1.5 $\times 10^3$	5429 878	47573 8669
2 $\times 10^3$	6496 991	56986 9793
2.5 $\times 10^3$	7752 1092	67484 10392
5 $\times 10^3$	12544 1231	109565 12493

Just like in the case of the Media Renderer, an exhaustive exploration ends up producing a very dense execution trace tree, which is hard to partition into

equivalence classes, due to the relatively high values of the relevant parameters: $m = 10$, and $\theta = 34$. This resulting dense trace tree brings, also in this case, the time complexity of the equivalence partitioning process close to its upper bound of $|\Sigma|^{3m}$, which for $|\Sigma| = 12$ and $m = 10$ become even more difficult to handle than in the Media Renderer case study. Thus, also in the case of the Light Control System component, to avoid having a very expensive equivalence partitioning phase, the number of learning traces is again severely restricted, and the trace tree is only partially filled.

Therefore, as it can been seen in table 5.11, the number of executions used to learn the Light Control System component has been limited to 5000. This limitation make the cycle identification ideal rather unreachable for this case study. However, even in this situation, the incomplete model learning algorithm still allows us to discover safe, usable paths in the otherwise unknown behaviour of the Light Control System.

Light Control System – Cycle-Based Exploration

Similarly to the Media Renderer case study, the cycle-based exploration strategy also offers here an interesting improvement over the unoptimized exploration. Since the cycle-based strategy focuses the available executions upon the specified usage scenarios, the algorithm ends up learning models significantly smaller than the ones learned by the normal exploration strategy. This effect can be observed from the summarized statistical data in table 5.12, and by comparing these model sizes against the ones in table 5.11.

The specification of interesting cycles is presented below.

```
:[1]<lon?><lonok!>[2]<loff?><loffok!>

<lon?><lonok!>:[2]<dim?><dim3!>[3]<bright?><bright4!>

<lon?><lonok!><dim?><dim3!>:[3]<dim?><dim2!>
[4]<bright?><bright3!>

<lon?><lonok!><dim?><dim3!><dim?><dim2!>:[4]<dim?><dim1!>
<bright?><bright2!>

:[1]<lon?><lonok!>[2]<dim?><dim3!>[3]<loff?><loffok!>
```

```
:[1]<lon?><lonok!>[2]<dim?><dim3!>[3]<dim?><dim2!>[4]<loff?>
<loffok!>

:[1]<lon?><lonok!>[2]<dim?><dim3!>[3]<dim?><dim2!>[4]<dim?>
<dim1!><loff?><loffok!>
```

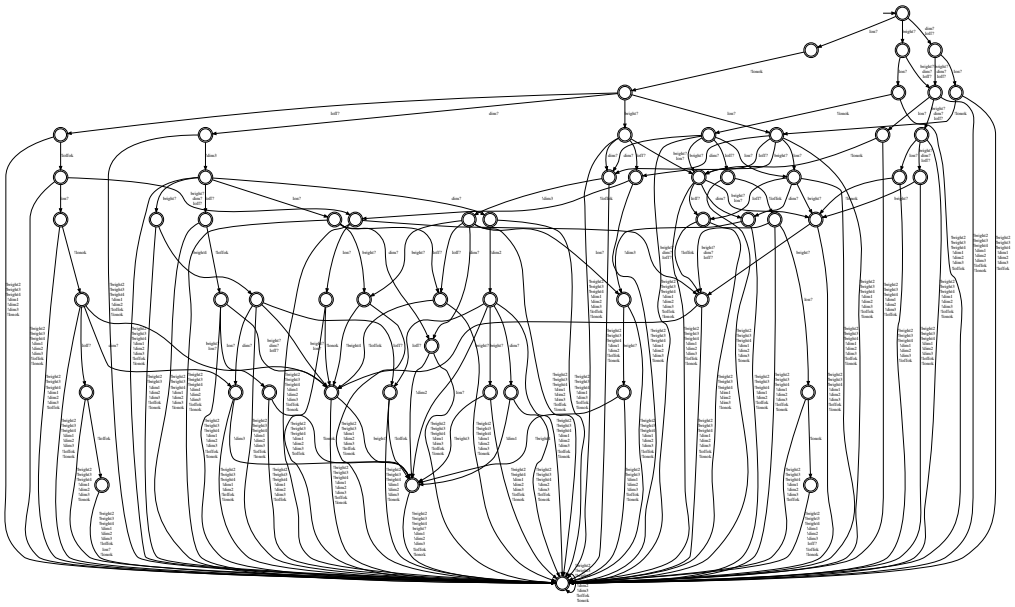
Table 5.12: Model inference results – normal exploration

nr.	LCS size	LCS tr.
200	841 59	7429 524
250	1021 58	9025 515
500	1695 62	15012 558
1 $\times 10^3$	1588 70	13903 584
1.5 $\times 10^3$	1607 68	14098 576
2 $\times 10^3$	2057 73	18106 623
2.5 $\times 10^3$	785 76	6737 511
5 $\times 10^3$	1207 72	10462 493

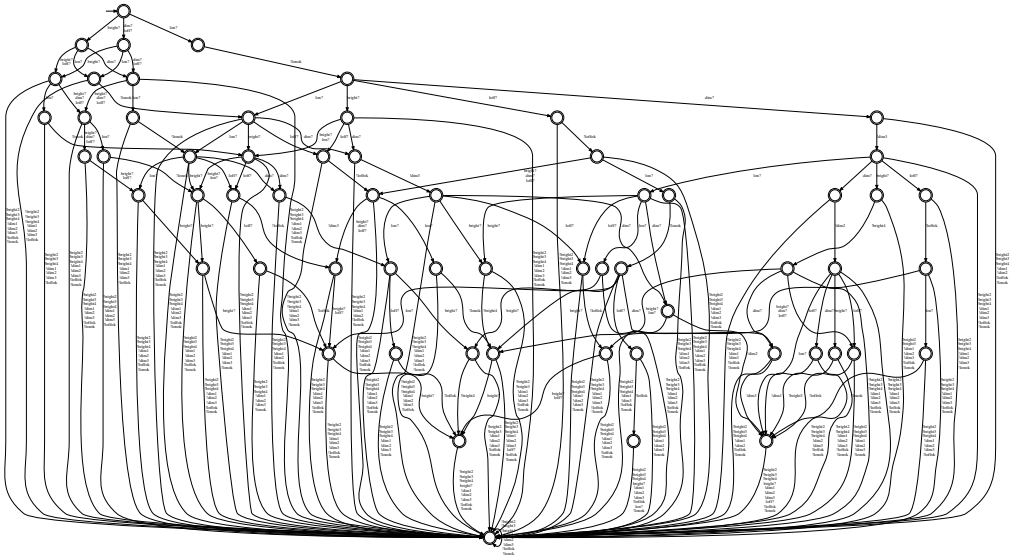
Some of the models obtained during the cycle-optimized learning are presented. As a significant difference between the Light Control System and the Media Renderer component, we have to mention that the Light Control System is apparently less controllable than the Media Renderer, having 4, instead of 5 controllable events, and 8, instead of 5, uncontrollable events, the effect of the cycle-based optimization strategy should be less visible here. Theoretically, it is harder to force a current run towards a desired path when you have more uncontrollable events, however, as this particular black box is output deterministic, the latter does not apply.

In figure 5.65(a) we can see the model of the Light Control System component obtained in 250 learning executions, while figure 5.65(b) presents a model learned in 500 executions. Further on, in figure 5.66(a) we can see a model of the Light

Control System learned in 2500 executions, while the model in figure 5.66(b) has been learned in 5000 executions.

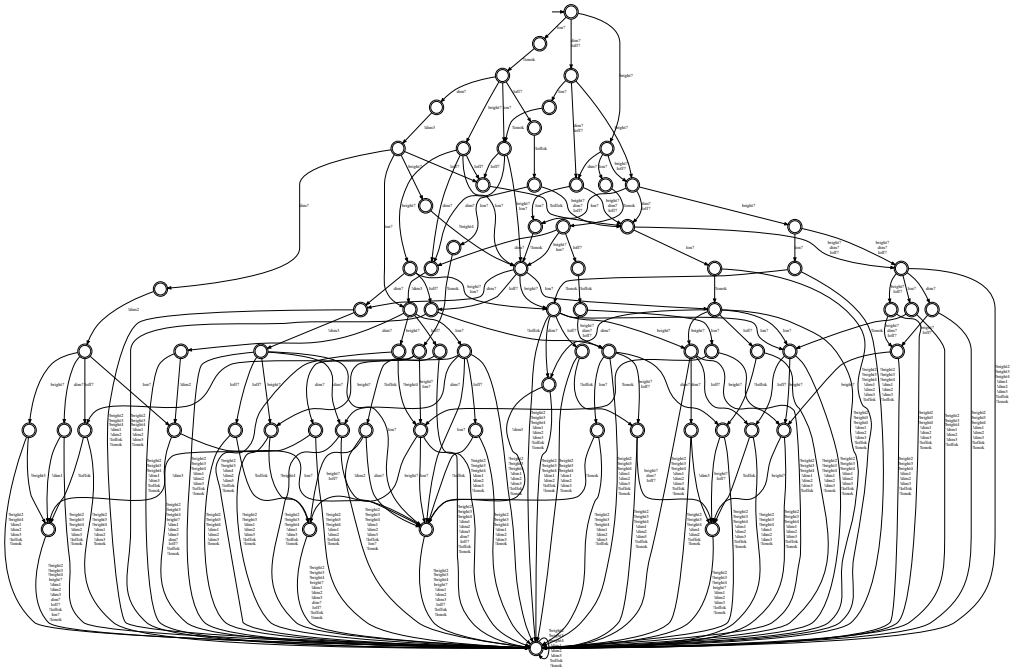


(a)

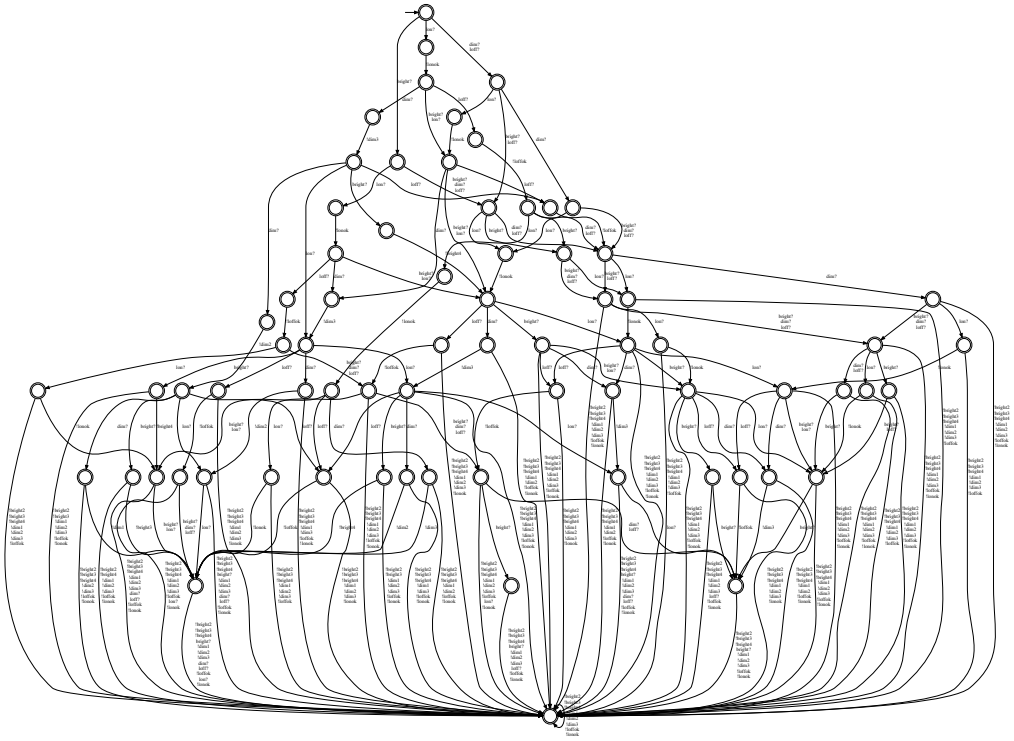


(b)

Figure 5.65: Learned models for component Light Control System in: (a) 250 executions, (b) 500 executions.



(a)



(b)

Figure 5.66: Learned models for component Light Control System in: (a) 2500 executions, (b) 5000 executions.

Although the execution has been always steered towards the relevant scenarios described in the cycle specification file, none of these desired loops has been identified within the maximum 5000 executions available. However, the models obtained by focusing on the relevant scenarios are significantly smaller than their unoptimized correspondents, and also well explored considering the number of executions employed. Thus, the model obtained after 250 executions has well-explored states up to a depth of 3, the one learned in 500 executions already has some well-explored states also at depth 4, while the number of outgoing uncontrollable transitions to the unknown-future state decreases further more after 2500 and 5000 learning executions. The model obtained in 2500 runs has completely explored states at depths up to 6 and 7, while the one using 5000 learning executions already has such states at depths of 7 and even 8.

5.5 Learning Large Individual Components

Out of the existing model learning techniques, we consider the work of Aarts and Vaandrager [1, 2] as being most related to ours, since they:

- also learn components that communicate asynchronously, although the models only contain input-determined behaviour, i.e. a maximum of one outgoing send transition is allowed from each state
- have also conducted experiments where they've only learned a partial behaviour of the components, that conformed to a certain "learning purpose", which, in our case, can be the desired temporal property.

Being asynchronous, the case studies they've considered were easy to adapt to our model, by only adding the assumption of inner nondeterminism with respect to the transitions on send events.

In both case studies described in [1, 2] the behavioural models of the components are learned in isolation, so the set of messages to be forwarded to the component is supposed to be available to the inference engine. The controller obtained is a local controller, that would ensure that the component behaviour conforms to the desired property. The properties we have considered are the same to the learning purposes described by Aarts and Vaandrager. It should be mentioned here, however, that Aarts and Vaandrager only use the learning purpose to restrict the model inference effort to a subset of interesting behaviour, and they do not try to obtain a controller that would enforce that behaviour.

Also, while Aarts and Vaandrager have conducted experiments on the real systems under test, our experiments have been performed, for simplicity, on abstract simulations of those components. However, this aspect only has impact on the actual time cost of a learning query, and does not influence the total number of queries needed for model inference, or the length of a query.

5.6 Case Study: The Session Initiation Protocol (SIP)

The Session Initiation Protocol (SIP) is an application layer protocol [35] that is widely used for communication sessions such as voice and video calls over Internet Protocol, etc. The case study considered models the behaviour of a SIP server while communicating with a SIP client.

The server and the client communicate by exchanging messages, which are of two types: Request and Response. Due to the fact that both Request and Response messages can take different parameter values, the actual alphabet of SIP is quite complex, having a large set of input symbols, which is reduced by Aarts and Vaandrager in [2] by means of various abstraction techniques to a relevant subset of only 16 events.

The abstract alphabet of a server implementing the SIP protocol, used in this case study is the following:

- INVITE_ f?, INVITE_ l?: INVITE request messages are used to establish a session between agents, and they can either be final (INVITE_ f) or provisional (INVITE_ l)
- ACK_ f?, ACK_ l?: ACK request messages are used to confirm reliable message exchanges between agents, and they can also be final (ACK_ f) or provisional (ACK_ l)
- PRACK_ f?, PRACK_ l?: PRACK: Provisional Response Acknowledgment request messages, used to acknowledge provisional responses
- timeout_ f!, timeout_ l!: response message sent by the server to notify a final or provisional timeout
- 100_ f!, 100_ l!: 100 responses are used to let the client know that the server is trying to fulfill its request

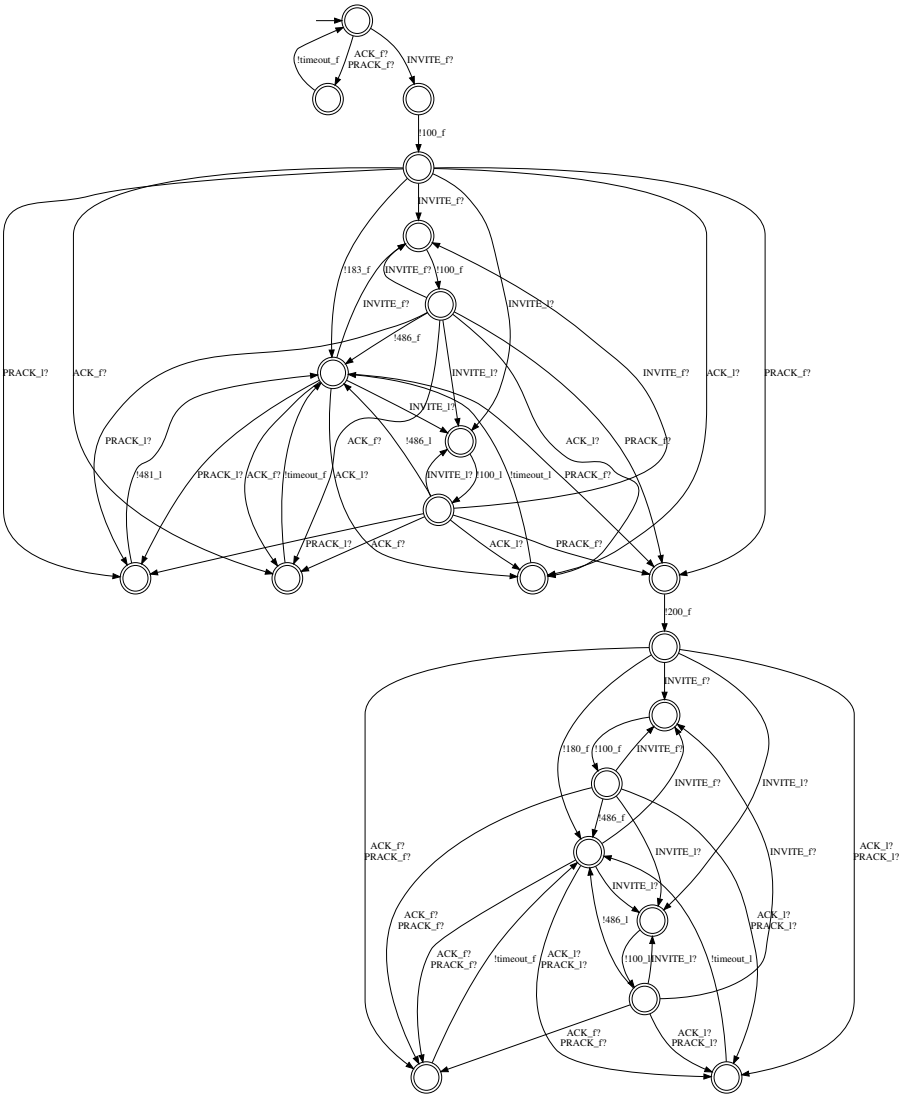
- 180_ f!: 180 responses mean that the destination user agent has received the INVITE message
- 183_ f!: 183 responses are used to send extra information during a session in progress
- 200_ f!: 200 response messages mean that the request was processed successfully
- 486_ f!, 486_ l!: 486 responses are used to let the client know that the server is busy
- 481_ l!: 481 responses are error messages meaning that the requested action doesn't exist

The real model of the Session Initiation Protocol, on which our simulated experiments are based, can be seen in figure 5.67(a), while the learning purpose automaton appears in figure 5.67(b). The learning purpose describes a behaviour in which only final request messages are allowed from the initial state, and where no two consecutive request messages are permitted. Further on, we have used the learning purpose automaton as the desired property automaton, to guide the behaviour exploration of the simulated black box component.

It is important to note here that, while the original model from [2] is input determined, the models considered by BASYL are not, thus allowing for more that one output transitions from the same state. This adds the issue of uncontrollable send transitions, but, since this model is more general than input determined I/O automata, behaviour that can be modelled by the latter can nevertheless be learned using the former.

However, while learning a complete model for an input-determined I/O automata, whose behaviour is entirely controllable, can be achieved using a version of the Angluin algorithm, doing the same in the presence of uncontrollability is more expensive. But, in contrast to the work in [2, 1], the purpose of BASYL is not inferring a complete model of the black box component/s under study, but a controllable approximation, that would allow us to enforce the desired safety property by generating a local controller.

Our of the 16 events of the SIP alphabet, only 6 are controllable, while the other 10 are uncontrollable. This leads to quite a large fairness bound for the SIP component: $\theta = 48$. The other relevant parameter, the longest size of an acyclic trace, is the largest of all considered case studies: $m = 17$. Thus, SIP is



(a)

(b)

Figure 5.67: (a) Session Initiation Protocol real model, (b) learning purpose

a very deep component compared to the rest, and thus hard to explore and to learn properly.

The learning experiments performed for the SIP component have only considered incomplete learning. Both the unoptimized exploration and the cycle-based exploration strategies have been experimented with.

Below, we provide the specification of the cycles considered for cycle-based exploration experiments. As the SIP component is significantly large and contains many relevant execution scenarios, with many loops and many join points of these loops, for convenience we have only extracted a small subset of those cycles to use in our experiments.

```
:[1]<ACK_f?><timeout_f!>

:[1]<PRACK_f?><timeout_f!>

<INVITE_f?><100_f!><ACK_f?>:[2]<timeout_f!>[6]<ACK_f?>

<INVITE_f?><100_f!><PRACK_l?>:[2]<481_l!>[6]<PRACK_l?>

<INVITE_f?><100_f!><INVITE_l?>:[4]<100_l!>[6]<INVITE_l?>

<INVITE_f?><100_f!><ACK_l?>:[5]<timeout_l!>[6]<ACK_l?>

<INVITE_f?><100_f!><PRACK_f?><200_f!><INVITE_l?>:[7]<100_l!>
<INVITE_l?>
```

Session Initiation Protocol – Normal Exploration

In the following, we present the results obtained in the learning process of the SIP component, when the normal exploration strategy is used. Table 5.13 summarizes the results of the incomplete model inference process. Both the learned model and the generated controller are presented, and for the learned model we present both the statistical data of the unminimized and of the minimized model. The computed controller is given only in its reduced form.

As the SIP component is a large one, having a relatively large alphabet size, a high fairness bound of $\theta = 48$ and an also large acyclic bound $m = 17$, SIP is not only hard to properly explore, but its equivalence partitioning is also hard,

due mainly to its large value for m. Thus, as the time cost of processing a nearly-full trace tree is prohibiting, we found it necessary, as in the Domotics case study, to decide for a low limit on the number of executions allowed, which then was set to 5000.

Table 5.13: Model inference results – normal exploration

nr.	SIP size	SIP tr.	ctrl size	ctrl tr.
200	2092 1025	21756 10710	1107	5605
250	2544 1160	26472 12139	1248	6364
500	4630 1686	48124 17667	1804	9348
1×10^3	8461 2288	88105 24329	10471	48330
1.5×10^3	11954	124382	14646	67181
2×10^3	15371	159872	18703	85466
2.5×10^3	18699	194628	22685	103376
5×10^3	33180	345338	39206	175933

The learned models, as it can be seen from table 5.13, are large, and they naturally increase in size as more and more new paths are explored. They are too large to be visualized and, starting with the results of the experiment employing 1500 executions, and further on, they are even too large to be minimized in a reasonable amount of time, i.e. less than an hour. For visualization, minimization and verification of the learned models, as for controller generation we have used the Supremica tool [59].

Session Initiation Protocol – Cycle-Based Exploration

The results of the cycle-based exploration for the SIP case study can be seen in table 5.14. As SIP is a large component in terms of the relevant parameters considered, and the number of learning executions has been limited to 5000,

it is not expected from the resulting models to have already identified cycles. The benefits of the cycle-oriented learning optimization can be seen, in this case, in the significantly smaller size of the minimized form of the obtained models. The first, and smallest of these models is presented in figure 5.68(a), while its corresponding controller can be seen in figure 5.68(b). Similarly to the unoptimized learning, the unminimized sizes of the learned models increases with the number of employed explorations, until their minimization becomes too expensive (starting with the 1500 executions bound).

Table 5.14: Model inference results – cycle-based exploration

nr.	SIP size	SIP tr.	ctrl size	ctrl tr.
200	2103 135	21760 1453	166	913
250	2567 155	26583 1686	186	1044
500	4625 254	47860 2726	297	1735
1 $\times 10^3$	8395 386	87057 4253	9778	44861
1.5 $\times 10^3$	12064	125014	13991	63902
2 $\times 10^3$	15364	159150	17644	80167
2.5 $\times 10^3$	18594	192763	21241	96152
5 $\times 10^3$	33003	342176	36822	164009

The large size difference between the two sets of models is also due to the fact that the specified cycles, used to guide the exploration, are only a small subset of the set of cycles of SIP. Thus, the learning executions have only focused on a small set of paths. These specified cycles are given again below.

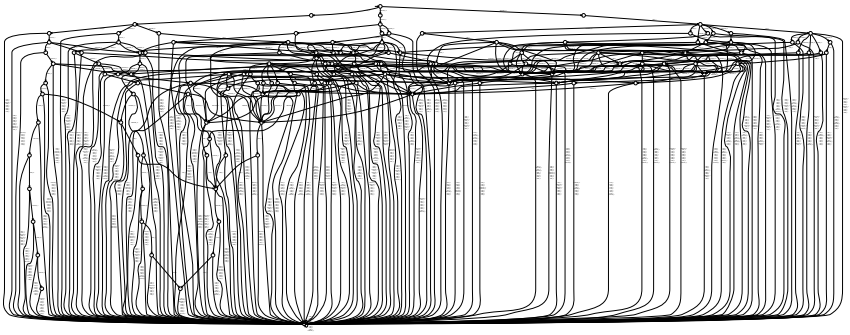
```
:[1]<ACK_f?><timeout_f!>

:[1]<PRACK_f?><timeout_f!>

<INVITE_f?><100_f!><ACK_f?>:[2]<timeout_f!>[6]<ACK_f?>
```

<INVITE_f?><100_f!><PRACK_l?>:[2]<481_l!>[6]<PRACK_l?>

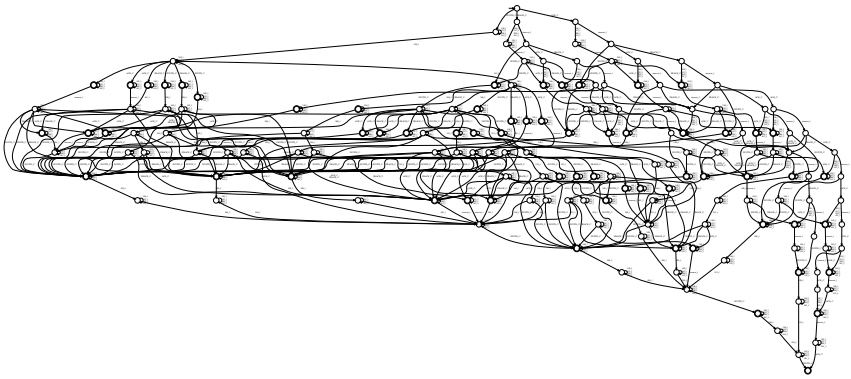
<INVITE_f?><100_f!><INVITE_l?>:[4]<100_l!>[6]<INVITE_l?>

<INVITE_f?><100_f!><ACK_l?>:[5]<timeout_l!>[6]<ACK_l?>

<INVITE_f?><100_f!><PRACK_f?><200_f!><INVITE_l?>:[7]<100_l!>
<INVITE_l?>



(a)



(b)

Figure 5.68: (a) Session Initiation Protocol learned model after 200 executions, (b) corresponding controller for the learning purpose property

Chapter 6

Conclusions

In this dissertation we have introduced a novel model learning method that makes it possible to automatically and reliably compose a system containing black box components with uncontrollable events. Our technique infers safe behavioural models, i.e. models that underapproximate real controllable behaviour and overapproximate uncontrollable behaviour, for the black box components.

An inferred model can be precise, if the component has relatively low values for its relevant parameters (alphabet size, fairness bound, maximum acyclic path) and an exhaustive exploration of its behaviour within cost limits is possible. If, however, the maximum number of allowed learning executions does not leave room for a complete exploration of the black box, the inferred models are safe approximations of the real component behaviour.

6.1 Summary of Contributions

The contributions of this work are in the field of formal methods and their applications to component-based software engineering. More specifically, it addresses a problem in the subfield of regular inference: learning safe approximate models for components with an alphabet that contains both controllable and uncontrollable events.

The main contribution brought by this work is a method for learning safe approximate models of black box components with uncontrollable events. This is

done using a limited number of allowed executions. Learning such safe approximations of real behaviour enables a safe composition for systems integrating asynchronous black box components with uncontrollable send events.

We have formally proved the correctness of the elaborated theoretical solutions. Also, in order to empirically validate our approach, we have developed specific tool support and conducted different experiments. We have implemented the developed learning techniques in the BASYL prototype and proved its usefulness in a set of case studies from both component-based and service-based software engineering literature.

Our main contribution, learning safe approximate models, has been realized by means of the following, lower-level contributions of this work:

- a centralized behaviour exploration method [51, 52, 53]: the behaviour of the system is globally explored in order to gather execution samples relevant to the composition goal. As the model of each black box component is inferred individually, the execution trace samples stored are actually projections of the global traces on each component's alphabet of events. Both positive and negative samples are considered. The execution is controlled by a proactive adaptor, which guides the current run towards the satisfaction of the system specification. The exploration process aims to cover as much as possible of the behaviour of each black box in part, but only under the constraint that each system execution trace conforms to the composition goal.

- a distributed behaviour exploration method [50, 54]: the behaviour of each component in the system is explored locally, together with the models of other components in the system, in order to allow for parallelism in the model inference process. In this case, each local execution is independently observed and controlled by a local proactive adaptor. Similarly to the centralized method, only traces that conform to the system specification are considered for exploration.

- a cycle-oriented optimization of the two behaviour exploration methods: as most use cases of composed systems require the repeated invocation of certain execution scenarios, we can make the most of a limited number of executions by focusing on cycle identification. This basically means that the exploration is prioritized for traces representing unrolled cycles. The cycles explored are the ones that can be found in the system specification automaton. If the number of cycles thus specified is too large,

the exploration focuses on a few relevant ones, corresponding to cyclic usage scenarios. This optimization results in an earlier cycle identification, smaller learned models and faster convergence of inferred automata to precise-like models.

- a model building method that makes use of all the runtime observations gathered during the exploration process, observations concerning both positive and negative execution samples. The model obtained represents a safe approximation of the real black box behaviour, that is the controllable behaviour of the component is underapproximated by the built model, while the uncontrollable behaviour of the component is overapproximated. This makes the inferred models of the black box components in the system suitable to be used in synthesizing a system adaptor that would realize the composition goal.

- the tool support called BASYL [53, 54], which we have developed and used for an experimental validation of our approach. It contains an implementation for each of the learning techniques described in this dissertation.

6.2 Limitations

The main limitation of our approach is, as discussed, its high complexity. First, learning a precise model for a black box component requires a high number of executions, however, we can provide a safe approximate model for any predetermined number of runs. Second, the cost of inferring safe approximations is a higher computational complexity as far as the model building process is considered, which can prove inadequate in certain cases. However, as usually the most expensive part of model learning is the actual interaction of the learner with the oracle [55], i.e. the black box, we consider the additional computational costs of model building a necessary harm.

Another important limitation of our approach is that the assumption that asynchronous black box components must acknowledge message receive events is a potentially strong constraint, and so is the assumption of bounded fairness with respect to the occurrence of uncontrollable events. However, for model learning to be possible in the context of uncontrollable behaviour, some constraining assumptions are needed. So far, both these assumptions are of vital importance to our learning technique.

Currently, our method works with manually derived abstractions of real components. Therefore, it needs an additional abstract-concrete mapping layer to be able to interact with real black box components, that have larger, more complex and possibly infinite alphabets. Such abstraction layers are developed for example in the work of Aarts and Vaandrager [2, 1], however, in the context of learning safe approximate models, the abstract alphabet obtained should also be a safe approximation of the real alphabet.

6.3 Future Work

As future directions of development, BASYL would benefit from rigorous extensions and optimizations that would address its limitations, improve its performances and release it from some of its current constraints.

The following directions are considered:

- a more efficient, possibly parallelized, equivalence partitioning method, in order to address the high complexity of the model building algorithm
- an adaptation of the behaviour exploration strategy to the case where no message receive acknowledgment is possible; finding efficient ways of deducing indirectly whether a forwarded message was accepted or rejected by a component
- developing automatic techniques for deriving safe abstractions for large, possibly infinite alphabets of events (for example, when messages with integer, real or string parameters are considered)
- an adaptation of both the behaviour exploration process and the model building algorithm to the case when the learning process starts from a partial, incomplete, potentially incorrect, or in some other way unreliable model; how would such an approach make use of the already existing information to increase the efficiency of learning safe approximations?

In the context of the latter considered extension, it is worth mentioning that, while our method does not depend on source code availability, it can be combined with static component interface extraction, which is known to produce overapproximated models, to provide more precise interfaces. Also, it can be applied to improve the knowledge on maintenance components for which no behavioural model has been provided, or provided models are outdated.

Appendix A

Publications

- Casandra Holotescu, *Local Model Learning for Asynchronous Services*, Proceedings of the 4th International Workshop on Principles of Engineering Service-Oriented Systems (PESOS 2012) – colocated with the 34th International Conference on Software Engineering (ICSE 2012), June 2012, Zurich, Switzerland [**IEEE Xplore, INSPEC, ISI circuit**] [54].
- Casandra Holotescu, *Asynchronous Learning for Service Composition*, 7th International Workshop on Engineering Service-Oriented Applications (WESOA 2011) – colocated with the 9th International Conference on Service Oriented Computing (ICSOC 2011), December 2011, Paphos, Cyprus, Lecture Notes in Computer Science, Springer-Verlag [**ACM, DBLP, INSPEC, ISI circuit**] [53].
- Casandra Holotescu, *Controlling the Unknown*, Preproceedings of the First International Conference on Verification of Object Oriented Systems (FoVeOOS 2010), June 2010, Paris, France [51].
  Cited by:
  - Jose Antonio Martin, Antonio Brogi and Ernesto Pimentel, *Learning from Failures: a Lightweight Approach to Run-Time Behavioural Adaptation*, Proceedings of the 8th International Symposium on Formal Aspects of Component Software (FACS 2011), Oslo, Norway, Lecture Notes in Computer Science, Springer-Verlag, 2011
  - Jose Antonio Martin, *Secure Adaptation of Software Services*, Ph.D. Thesis, Universidad de Màlaga, 2012

- Casandra Holotescu, *Error-avoiding Adaptors for Black box Software Components*, Doctoral Symposium of the 25th ACM/IEEE International Conference on Automated Software Engineering (ASE 2010), September 2010, Antwerp, Belgium [**ACM, DBLP**] [52].
- Casandra Holotescu, *Black Box Composition: a Dynamic Approach*, 9th International Workshop on Specification and Verification of Component Based Systems (SAVCBS 2010) – colocated with the 18th International Symposium on the Foundations of Software Engineering (FSE 2010), November 2010, Santa Fe, New Mexico, United States of America [50].

Appendix B

List of Abbreviations

ATL Alternate Time Logic

BASYL Black box Asynchronous Learning (Tool)

CAD Computer Aided Design

CBSE Component Based Software Engineering

FSM Finite State Machine

LCS Light Control System

LTL Linear Temporal Logic

LTS Labeled Transition System

MR Media Renderer

SIP Session Initiation Protocol

SSO Single Sign On (Protocol)

SSO CI Single Sign On (Protocol) Client

SSO SP Single Sign On (Protocol) Service Provider

SSO IDP Single Sign On (Protocol) Identity Provider

TT ThinkTeam

Bibliography

[1] Fides Aarts, Julien Schmaltz, and Frits Vaandrager. Inference and abstraction of the biometric passport. In *Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation - Volume Part I*, ISoLA'10, pages 673–686, 2010.

[2] Fides Aarts and Frits Vaandrager. Learning i/o automata. In *Proceedings of the 21st international conference on Concurrency theory*, CONCUR'10, pages 71–85, 2010.

[3] Raffaela Mirandola Andreas Rausch, Ralf Reussner and František Plášil, editors. *The Common Component Modeling Example*, volume LNCS 5153 of *Lecture Notes in Computer Science*. Springer Berlin/Heidelberg, 2008.

[4] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87 – 106, 1987.

[5] Marco Autili, Paola Inverardi, Alfredo Navarra, and Massimo Tivoli. Synthesis: A tool for automatically assembling correct and distributed component-based systems. In *Proc. 29^{th} Int'l. Conf. on Software Engineering*, pages 784–787, 2007.

[6] Marco Autili, Paola Inverardi, and Massimo Tivoli. Automatic adaptor synthesis for protocol transformation. In *Proceedings of the First International Workshop on Coordination and Adaptation Techniques for Software Entities*, 2004.

[7] Philippe Balbiani, Fahima Cheikh, and Guillaume Feuillade. Algorithms and complexity of automata synthesis by asynhcronous orchestration with applications to web services composition. *Electronic Notes in Theoretical Computer Science*, 229(3):3 – 18, 2009. Proceedings of the First Interaction and Concurrency Experiences Workshop (ICE 2008).

[8] Hubert Baumeister, Florian Hacklinger, Rolf Hennicker, Alexander Knapp, and Martin Wirsing. A component model for architectural programming. *Electronic Notes in Theoretical Computer Science*, 160:75 – 96, 2006. Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005).

[9] Steffen Becker, Sven Overhage, and Ralf H. Reussner. Classifying software component interoperability errors to support component adaption. In *Component-Based Software Engineering, 7th International Symposium, CBSE 2004*, pages 68–83. Springer, 2004.

[10] Nikola Beneš, Ivana Černá, and Filip Štefaňák. Factorization for component-interaction automata. In Mária Bieliková, Gerhard Friedrich, Georg Gottlob, Stefan Katzenbeisser, and Gyoergy Turán, editors, *SOFSEM 2012: Theory and Practice of Computer Science*, volume 7147 of *Lecture Notes in Computer Science*, pages 554–565. Springer Berlin / Heidelberg, 2012.

[11] Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines using domains with equality tests. In JosÃľ Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering*, volume 4961 of *Lecture Notes in Computer Science*, pages 317–331. Springer Berlin / Heidelberg.

[12] Antonia Bertolino, Paola Inverardi, Patrizio Pelliccione, and Massimo Tivoli. Automatic synthesis of behavior protocols for composable web-services. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 141–150, 2009.

[13] David Blevins. Component-based software engineering. chapter Overview of the Enterprise Java Beans Component Model, pages 589–606. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[14] Therese Bohlin, Bengt Jonsson, and Siavash Soleimanifard. Inferring compact models of communication protocol entities. In *Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings*, pages 658–672, 2010.

[15] Benedikt Bollig, Peter Habermehl, Carsten Kern, and Martin Leucker. Angluin-style learning of nfa. In *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI'09, pages 1004–1009, 2009.

[16] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Daniel Neider, and David R. Piegdon. Libalf: the automata learning framework. In *Proceedings of the 22nd international conference on Computer Aided Verification*, CAV'10, pages 360–364, 2010.

[17] Andrea Bracciali, Antonio Brogi, and Carlos Canal. Systematic component adaptation,. *Electronic Notes in Theoretical Computer Science*, 66(4), 2002.

[18] Andrea Bracciali, Antonio Brogi, and Carlos Canal. A formal approach to component adaptation. *J. Syst. Softw.*, 74(1):45–54, 2005.

[19] Gready Brooch. Addison-Wesley, Washington, 1993.

[20] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Qu´ ema, and Jean-Bernard Stefani. The fractal component model and its support in java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006.

[21] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, SERA '06, pages 40–48, Washington, DC, USA, 2006. IEEE Computer Society.

[22] Carlos Canal, Juan Manuel, and Murillo Pascal Poizat. Software adaptation. In *in L'objet, 12(1):9-31, 2006. Special Issue on Coordination and Adaptation Techniques for Software Entities*, pages 9–31.

[23] Carlos Canal, Pascal Poizat, and Gwen SalaÃijn. Model-based adaptation of behavioral mismatching components. *IEEE Transactions on Software Engineering*, 34(4):546–563, 2008.

[24] Luca Cavallaro, Elisabetta Di Nitto, Patrizio Pelliccione, Matteo Pradella, and Massimo Tivoli. Synthesizing adapters for conversational web-services from their wsdl interface. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, pages 104–113, 2010.

[25] Chia Yuan Cho, Domagoj Babić, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. MACE: Model-inference-Assisted

Concolic Exploration for Protocol and Vulnerability Discovery. In *Proceedings of the 20th USENIX Security Symposium*, Aug 2011.

[26] T.S. Chow. Testing software design modelled by finite state machines. *IEEE Transactions on Software Engineering*, pages 178–187, 1978.

[27] Ivica Crnkovic. *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA, 2002.

[28] Ivica Crnkovic, Michel Chaudron, and Stig Larsson. Component-based development process and component lifecycle. In *Proceedings of the International Conference on Software Engineering Advances*, ICSEA '06, pages 44–, Washington, DC, USA, 2006. IEEE Computer Society.

[29] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. Generating test cases for specification mining. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 85–96, 2010.

[30] Rami Marelly David Harel, Hillel Kugler and Amir Pnueli. Smart play-out. OOPSLA '03, October 2003. demo paper.

[31] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 109–120, New York, NY, USA, 2001. ACM.

[32] Nachum Dershowitz, editor. *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, volume 2772 of *Lecture Notes in Computer Science*. Springer, 2003.

[33] M. Yannakakis Doron Peled, Moshe Y. Vardi. Black box checking. In *In FORTE/PSTV*, pages 225–240. Kluwer, 1999.

[34] Khaled El-Fakih, Roland Groz, Muhammad Naeem Irfan, and Muzammil Shahbaz. Learning finite state models of observable nondeterministic systems in a testing context. In *22nd IFIP International Conference on Testing Software and Systems*, pages 97–102, Natal, Brazil, 2010.

[35] J. Rosenberg et al. *SIP: Session Initiation Protocol*. Network Working Group, June 2002.

[36] N. Ragouzis et al. *Security Assertion Markup Language (SAML) V2.0 Technical Overview*. OASIS Committee, March 2008.

[37] Tim Ewald. Component-based software engineering. chapter Overview of COM+, pages 573–588. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[38] Charles Buhman Santiago Comella-Dorda Fred Long John Robert Robert Seacord Felix Bachmann, Len Bass and Kurt Wallnau. Technical concepts of component-based software engineering, 2nd edition. Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, May 2000.

[39] Jerry Zayu Gao, Jacob Tsao, Ye Wu, and Taso H.-S. Jacob. *Testing and Quality Assurance for Component-Based Software*. Artech House, Inc., Norwood, MA, USA, 2003.

[40] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *In Proceedings of the 17th International Conference on Software Engineering*, pages 179–185, 1995.

[41] Carlo Ghezzi, Andrea Mocci, and Mattia Monga. Synthesizing intensional behavior models by graph transformation. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 430–440, Washington, DC, USA, 2009. IEEE Computer Society.

[42] Dimitra Giannakopoulou, Corina S. Păsăreanu, and Howard Barringer. Component verification with automatically generated assumptions. *Automated Software Engg.*, 12(3):297–320, 2005.

[43] Immo Grabe, Marcel Kyas, Martin Steffen, and Arild Torjusen. Executable interface specifications for testing asynchronous creol components. In Farhad Arbab and Marjan Sirjani, editors, *Fundamentals of Software Engineering*, volume 5961 of *Lecture Notes in Computer Science*, pages 324–339. Springer Berlin / Heidelberg.

[44] Roland Groz, Keqin Li, Alexandre Petrenko, and Muzammil Shahbaz. Modular system verification by inference, testing and reachability analysis. In *Proceedings of the 20th IFIP TC 6/WG 6.1 international conference on Testing of Software and Communicating Systems: 8th International Workshop*, TestCom '08 / FATES '08, pages 216–233, 2008.

[45] Mary Jean Harrold, Donglin Liang, and Saurabh Sinha. An approach to analyzing and testing component-based systems. In *ICSE 99*, May 1999.

[46] Wilhelm Hasselbring. Component-based software engineering. *International Journal of Software Engineering and Knowledge Engineering*, 2002.

[47] Michi Henning. The rise and fall of corba. *ACM Queue*, 4(5):28–34, June 2006.

[48] Kunihiko Hiraishi. Synthesis of supervisors using learning algorithm of regularlanguages. *Discrete Event Dynamic Systems*, 11:211–234, July 2001.

[49] Florian Hoelzl and Martin Feilkas. Autofocus 3 - a scientific tool prototype for model-based development of component-based, reactive, distributed systems. In Holger Giese, Gabor Karsai, Edward Lee, Bernhard Rumpe, and Bernhard Schaetz, editors, *Model-Based Engineering of Embedded Real-Time Systems*, volume 6100 of *Lecture Notes in Computer Science*, pages 317–322. Springer Berlin / Heidelberg, 2011.

[50] Casandra Holotescu. Black-box composition: a dynamic approach. In *9th International Workshop on Specification and Verification of Component Based Systems*, November 2010.

[51] Casandra Holotescu. Controlling the unknown. In *Preproceedings of the First International Conference on Verification of Object Oriented Software*, pages 283–297, June 2010. ISSN: 2190-4782.

[52] Casandra Holotescu. Error-avoiding adaptors for black-box software components. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, pages 487–492. ACM, 2010. Doctoral Symposium.

[53] Casandra Holotescu. Asynchronous learning for service composition. In *Proceedings of the 7th International Workshop on Engineering Service-Oriented Applications (WESOA 2011)*, volume 7221 of *Lecture Notes in Computer Science*, pages 76–88. Springer-Verlag Berlin Heidelberg, December 2011. ISBN: 978-3-642-31874-0, 978-3-642-31875-7, ISSN: 0302-9743.

[54] Casandra Holotescu. Local model learning for asynchronous services. In *Proceedings of the 4th International Workshop on Principles of Engineering Service-Oriented Systems (PESOS 2012)*, pages 925–931. IEEE press, June 2012. ISBN: 978-1-4673-1754-2, ISSN: 2156-7921.

[55] Falk Howar, Bernhard Steffen, and Maik Merten. From zulu to rers. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, Lecture Notes in Computer Science, pages 687–704. Springer Berlin / Heidelberg, 2010.

[56] He Jifeng, Xiaoshan Li, and Zhiming Liu. Component-based software engineering. In Dang Van Hung and Martin Wirsing, editors, *Theoretical Aspects of Computing – ICTAC 2005*, volume 3722 of *Lecture Notes in Computer Science*, pages 70–95. Springer Berlin / Heidelberg, 2005.

[57] Kathrin Kaschner. Conformance testing for asynchronously communicating services. In Gerti Kappel, Zakaria Maamar, and Hamid R. Motahari-Nezhad, editors, *9th International Conference on Service Oriented Computing, ICSOC 2011, December 5-8, 2011, Paphos, Cyprus, Proceedings*, volume 7084 of *Lecture Notes in Computer Science*, pages 108–124. Springer-Verlag, December 2011.

[58] M. J. Kearns and U. V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.

[59] Hugo Flordal Robi Malik Knut AAkesson, Martin Fabian. Supremica âĂŞ an integrated environment for verification, synthesis and simulation of discrete event systems. In Ann Arbor, editor, *Proceedings of 2006 Workshop of Discrete Event Systems (WODES)*, Michigan, 2006.

[60] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 501–510, 2008.

[61] O. Maler and A. Pnueli. On the learnability of infinitary regular sets. *Information and Computation*, 118(2):316 – 326, 1995.

[62] M. Douglas McIlroy. Mass-produced software components. In J. M. Buxton, Peter Naur, and Brian Randell, editors, *Software Engineering Concepts and Techniques (1968 NATO Conference of Software Engineering)*, pages 88–98. NATO Science Committee, oct 1968.

[63] Bertrand Meyer. Prentice Hall International Series in Computer Science, New York: Prentice-Hall, 1988.

[64] Oliver Niese. *An Integrated Approach to Testing Complex Systems*. PhD thesis, University of Dortmund, 2003.

[65] Harald Raffelt, Bernhard Steffen, Therese Berg, and Tiziana Margaria. Learnlib: a framework for extrapolating behavioral models. *Int. J. Softw. Tools Technol. Transf.*, 11:393–407, October 2009.

[66] Prabhakar Raghavan Rajeev Motwani. *Randomized algorithms*, chapter 3.6. The Coupon Collector's Problem, pages 57–63. Number MR1344451. Cambridge University Press, Cambridge, 1995.

[67] P.J. Ramadge and W.M. Wonham. The control of discrete event systems. *Proc. of the IEEE*, 77(1), January 1989.

[68] Ralf H. Reussner. Adapting Components and Predicting Architectural Properties with Parameterised Contracts. In Wolfgang Goerigk, editor, *Tagungsband des Arbeitstreffens der GI Fachgruppen 2.1.4 und 2.1.9, Bad Honnef*, pages 33–43, 2001.

[69] Ralf H. Reussner. The Use of Parameterised Contracts for Architecting Systems with Software Components. In Wolfgang Weck, Jan Bosch, and Clemens Szyperski, editors, *Proceedings of the Sixth International Workshop on Component-Oriented Programming (WCOP'01)*, June 2001.

[70] R. L. Rivest and R. E. Shapire. Inference of finite automata using homing sequences. In *Machine Learning: From Theory to Applications*.

[71] Heinz W. Schmidt and Ralf H. Reussner. Automatic component adaptation by concurrent state machine retrofitting. Technical report, Fakultat fur Informatik, Universitat Karlsruhe, Am Fasanengarten 5, D-76128, 2000.

[72] Heinz W. Schmidt and Ralf H. Reussner. Generating adapters for concurrent component protocol synchronisation. In *Proc. 5^{th} IFIP Int'l. Conf. on Formal Methods for Open Object-Based Distributed Systems*, pages 213–229, 2002.

[73] R. Seguel, R. Eshuis, and P. Grefen. An overview on protocol adaptors for service component integration. In *WP265*. Eindhoven University of Technology, December 2008.

[74] Muzammil Shahbaz. *Reverse Engineering Enhanced State Models of Black Box Components to support Integration Testing*. PhD thesis, Grenoble Institute of Technology, 2008.

[75] Ian Sommerville. *Software Engineering 8th edition*. Addison Wesley, 2007.

[76] Rajiv Ranjan Suman, Rajib Mall, Srihari Sukumaran, and Manoranjan Satpathy. Extracting state models for black-box software components. *Journal of Object Technology*, 9(3):79–103, May 2010.

[77] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[78] Massimo Tivoli. *An architectural approach to the automatic composition and adaptation of software components*. PhD thesis, Universita di L'Aquila, 2005.

[79] M.P. Vasilevskii. Failure diagnosis of automata. *Kibernetika*, pages 98–108, 1973.

[80] Nanbor Wang, Douglas C. Schmidt, and Carlos O'Ryan. Component-based software engineering. chapter Overview of the CORBA component model, pages 557–571. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[81] W.M. Wonham. *Supervisory control of discrete event systems*. Systems Control Group, Dept. of Electrical and Computer Engineering, University of Toronto, 2009.

[82] Gaoyan Xie and Zhe Dang. Ctl model-checking for systems with unspecified finite state components. In *Proceedings of the 3rd workshop on specification and verification of component-based systems (SAVCBS'04), affiliated with ACM SIGSOFT 2004/FSE-12*, pages pp. 32–38, 2004.

[83] Nikola; Kostic Dejan; Kuncak Viktor Yabandeh, Maysam; Knezevic. Crystalball: Predicting and preventing inconsistencies in deployed distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)*, April 2009.

[84] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333, 1997.