THÈSE DE DOCTORAT
DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

Présentée par

Madame Aiswarya CYRIAC

Pour obtenir le grade de

DOCTEUR DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

Domaine :
Informatique

Sujet de la thèse :

Verification of Communicating Recursive Programs via Split-width

Thèse présentée et soutenue à Cachan le 28/01/2014 devant le jury composé de :

ABDULLA, Parosh Aziz	Professeur	Rapporteur
BOLLIG, Benedikt	Charge de Recherche	Co-directeur de thèse
BOUAJJANI, Ahmed	Professeur	Examinateur
GASTIN, Paul	Professeur	Directeur de thèse
MUSCHOLL, Anca	Professeur	Rapporteuse
WORRELL, James	Professeur	Examinateur

Laboratoire Spécification et Vérification
ENS de Cachan, UMR 8643 du CNRS
61, avenue du Président Wilson
94235 CACHAN Cedex, France

Abstract

This thesis investigates automata-theoretic techniques for the verification of physically distributed machines communicating via unbounded reliable channels. Each of these machines may run several recursive programs (multi-threading). A recursive program may also use several unbounded stack and queue data-structures for its local-computation needs. Such real-world systems are so powerful that all verification problems become undecidable.

We introduce and study a new parameter called split-width for the under-approximate analysis of such systems. Split-width is the minimum number of splits required in the behaviour graphs to obtain disjoint parts which can be reasoned about independently. Thus it provides a divide-and-conquer approach for their analysis. With the parameter split-width, we obtain optimal decision procedures for various verification problems on these systems like reachability, inclusion, etc. and also for satisfiability and model checking against various logical formalisms such as monadic second-order logic, propositional dynamic logic and temporal logics.

It is shown that behaviours of a system have bounded split-width if and only if they have bounded clique-width. Thus, by Courcelle's results on uniformly bounded-degree graphs, split-width is not only sufficient but also necessary to get decidability for MSO satisfiability checking.

We then study the feasibility of distributed controllers for our generic distributed systems. We propose several controllers, some finite state and some deterministic, which ensure that the behaviours of the system have bounded split-width. Such a distributedly controlled system yields decidability for the various verification problems by inheriting the optimal decision procedures for split-width. These also extend or complement many known decidable subclasses of systems studied previously.

Résumé

Cette thèse développe des techniques base d'automates pour la vérification formelle de systèmes physiquement distribués communiquant via des canaux fiables de tailles non bornées. Chaque machine peut exécuter localement plusieurs programmes récursifs (multi-threading). Un programme récursif peut également utiliser pour ses calculs locaux des structures de données non bornées, comme des files ou des piles. Ces systèmes, utilisés en pratique, sont si puissants que tous leurs problèmes de vérification deviennent indécidables.

Nous introduisons et étudions un nouveau paramètre, appelé largeur de coupe (split-width), pour l'analyse de ces systèmes. Cette largeur de coupe est définie comme le nombre minimum de scissions nécessaires pour partitioner le graphe d'une exécution en parties sur lesquelles on pourra raisonner de manière indépendante. L'analyse est ainsi réalisée avec une approche diviser pour régner. Lorsqu'on se restreint la classe des comportements ayant une largeur de coupe bornée par une constante, on obtient des procédures de décision optimales pour divers problèmes de vérification sur ces systèmes tels que l'accessibilité, l'inclusion, etc. ainsi que pour la satisfaisabilité et le model checking par rapport divers formalismes comme la logique monadique du second ordre, la logique dynamique propositionnelle et des logiques temporelles.

On montre aussi que les comportements d'un système ont une largeur de coupe bornée si et seulement si ils ont une largeur de clique bornée. Ainsi, grâce aux résultats de Courcelle sur les graphes de degré uniformément borné, la largeur de coupe est non seulement suffisante, mais aussi nécessaire pour obtenir la décidabilité du problème de satisfaisabilité d'une formule de la logique monadique du second ordre.

Nous étudions ensuite l'existence de contrôleurs distribués génériques pour nos systèmes distribués. Nous proposons plusieurs contrôleurs, certains ayant un nombre fini d'états et d'autres étant déterministes, qui assurent que les comportements du système sont des graphes ayant une largeur de coupe bornée. Un système ainsi contrôlé de manière distribuée hérite des procédures de décision optimales pour les différents problèmes de vérification lorsque la largeur de coupe est bornée. Cette classe décidable de système généralise plusieurs sous-classes décidables étudiées précédemment.

Acknowledgement

Paul Gastin, my mentor and role model, has also been a great friend throughout my stay in France. I have learned a lot from him – from mathematics, to doing research, to writing proofs, to doing sports, to appreciating nature and life! I express my deepest gratitude to him for all what I have learned from him, and for all the patience, support and confidence he had for me.

I express my heartfelt gratitude to Benedikt Bollig, my co-advisor, for his unfailing support, patience, encouragement and kindness throughout my doctoral studies and especially during the writing of this manuscript.

I would like to thank K. Narayan Kumar for the many pleasant research discussions, and also for making my stay in CMI very enjoyable. I also thank him for his inspiring course on verification when I was a master student at IMSc.

I thank my other co-authors Loïc Hélouët, Ahmed Kara, Thomas Schwentick and Marc Zeitoun for many insightful discussions. I particularly appreciate the discussions with Thomas Schwentick, and his words of encouragement. I also had the chance to work with Siddarth Krishna, I thank him for the many enjoyable discussions.

I thank DIGITEO for funding my PhD studies through its project LoCoReP, and LIA Informel, ENS Cachan, CNRS and INRIA for supporting many research visits and travels.

LSV has offered a very friendly and productive atmosphere. The administrative staff have been very helpful, especially Catherine Forestier, Virginie Guenard, Thida Iem of LSV and Sripathy of CMI. Special thanks to Bogdana Neuville for the help with visa and residence permit.

I would like to thank all my friends. My officemates Benjamin Monmege and Benoît Barbot were of immense help when it comes to French language and technical issues. Benjamin (Monfrère) has been virtually another supervisor for my thesis. Without him, I would not have survived the last four years. Many thanks to Christoph Haase, Benedikt Bollig, and Paul Gastin for the numerous coffee-breaks that I enjoyed a lot. Many thanks to Prakash Saivasan and Prateek Karandikar for the dinners, discussions and photographs. I also thank Olga Kovalova, Renuka Tayade, Jennifer Thomas, Suman Das, Rajarshi Pal, Meera Mohan, Nimisha James, Ocan Sankur, Cesar Rodriguez, Sandie Balaguer, Mahsa Shirmohammadi, Laure Daviaud, Eleni-Maria Vretta and Sinem Kavak for the many enjoyable moments and chats that we had.

I am eternally indebted to the Gastin family, for the immense kindness they showed to me, for offering me a home, for offering me a family. I thank one and all of you, Paul, Hélène, Maxime, Alice, Mélodie, and Mémé.

I thank my parents for their unceasing prayers for my studies. I am deeply grateful to my brother for his unconditional support and inspiration. All my love goes to Amal.

Aiswarya Cyriac
Cachan, France
January 2014

Contents

Part I

Concurrent Processes with Data-Structures

Chapter 1

Introduction

Formal verification has been a significant area of research in the recent years. The main reason is the vital role that technology plays today – almost everything one interacts with is directly or indirectly controlled by programs. A hardware or software failure may affect the quality of life of millions, or more seriously, cost lives, time and money. The necessity for "reliable" hardware and software is ever more prominent, and formal verification is an important tool in achieving this goal.

When verification as a whole turns out to be infeasible, one tries to identify the features she wants to verify. For instance, a crucial feature which needs to be verified may be the locking mechanism of a multi-threaded program. In another scenario, the timing delays and its propagation might be the crucial issue. In general, abstracting away several details and concentrating on particular features can make verification "feasible". Since there are several levels and dimensions of abstraction, formal verification has also several dimensions.

This thesis is a humble attempt to understand and extend this research area considering concurrency and recursion as the main features. It adopts the model checking approach for verification. We abstract away many details like timing delays and probabilities. However we keep concurrency, the power of recursion and communication between concurrent systems. We study verification of properties expressed in powerful logical formalism such as Monadic Second Order Logic (MSO), in addition to the ubiquitous control state reachability problem.

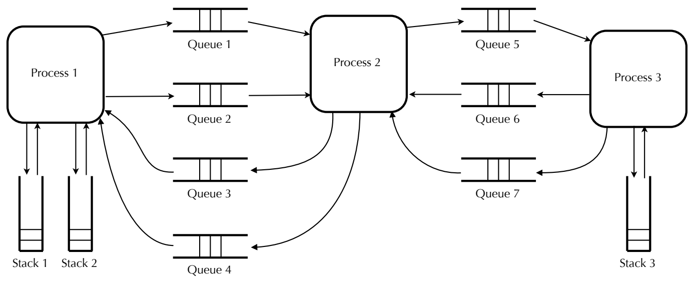
1.1 Thesis in a nutshell

The work described in this thesis is done in collaboration with Paul Gastin and K. Narayan Kumar. It is a generalisation of [CGN12a] where split-width is introduced to address the decidability of MSO specifications for multi-pushdown systems.

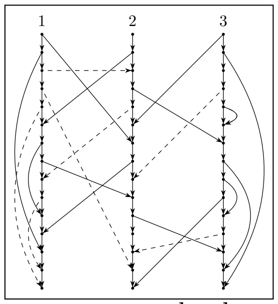
The study of communicating concurrent systems is an important and challenging research area. Various means of interaction between the concurrent processes have been considered by researchers, of which two prominent ones are 'shared-variable communication' and 'communication via channels'.

Different processes running on the same machine may employ 'shared-variable communication' as the main method of interaction, while physically distributed programs rely on communication channels. The former corresponds to 'synchronous communication' while the latter gives rise to 'asynchronous communication'.

We will consider generic systems which incorporate both these methods of communication (cf. see the figure below). We will be considering physically distributed machines which communicate via (possibly several) reliable first-in-first-out queues. Each of these machines are capable of running potentially recursive multi-threaded programs. These programs within a machine use shared variable for communication. Such a machine consisting of a set of threads communicating by shared memory can be formally modelled as a multi-pushdown system. Thus we have a network of multi-pushdown systems communicating via FIFO queues. Moreover, these programs may use stacks and queues as data-structures to aid their local computation. We call such a system a system of concurrent processes with data-structures (CPDS).



This models real-world systems like several computers connected via a local-area-network. Each of these computers may run several recursive programs concurrently, which may communicate among themselves via shared memory access. A computer may also provide several data-structures to the programs for the storage and retrieval of data. These computers may access the network to communicate with its peers, via first-in-first-out channels.



The behaviours of such system may be modelled by unifying message sequence charts [IT11] and nested-words [AM09]. We call them MSCNs. The behaviour of each process can be seen as a word, and hence an MSCN is a union of linear orders. We abstract the communication and the recursion by linking pairs of matching writes and reads to a data-structure. An example is depicted on the right.

The generic system of CPDS, when restricted to one process and only stack data-structures yields the trending topic of multi-pushdown systems [QR05, LMP07, BCCCR96, AKS13, LN11, ABH08]. On the other hand, restricting CPDS to have at most one queue between every pair of processes and no stacks gives us the well studied model of communicating finite state machines [BZ83, GKM06, GKM07, HMN^{+}05].

For specifying properties of CPDS we may consider temporal logics, or path expressions or monadic second order logic (MSO). These logics can easily link matching writes and reads and hence are very expressive.

Since CPDS are Turing powerful, verification of even basic properties like reachability becomes undecidable. However, the verification of these systems is an important concern. One way around this difficulty is to verify *approximations* of such systems. That is, identify a subset of the behaviours for which the verification problem becomes decidable. Naturally it is better if the subset covers many, if not most, or all of the interesting and substantial behaviours of the system. In practice bugs are likely to manifest among such behaviours.

Identifying such subsets in a parametrised way so that one may increase the coverage of the set by such a parameter would clearly be useful and all the more so if every behaviour is covered by some choice of the parameter. Yet another desired property for such a subclass is, if one may force the system in some operational way to only behaviours defined in the subset, this will give the user or implementer the choice of using a verified system at the cost of generality. The main contribution of this thesis is a systematic way to construct such approximations.

We introduce and study a new technique called split-width for the under-approximate verification of CPDS. This parameter is based on simple shuffle and merge operations and gives us a divide-conquer-way to prove the bound of languages. When parametrised by a bound on split-width, we obtain decidability for various verification problems. We provide a uniform decision procedure for various verification problems with optimal complexities.

We expose the power of split-width in several ways. We show that our simple algebra is powerful enough to capture any class of CPDS which admits decidability for MSO model checking, and yardstick graph metrics such as tree-width and clique-width.

We also show that various restrictions well-studied in the literature for obtaining decidability of reachability for the particular cases of multi-pushdown systems and message passing systems admit a bound on split-width. In fact, we propose generic controllers which subsume many of these cases.

Distributed controller design amounts to designing a controller (which is another CPDS) which, when run synchronously with a system ensures bounded split-width. These controllers are distributed in nature and are independent of the system it is controlling. Thus such a controller respects the privacy of the system (by not reading their states, for instance). Moreover, thanks to split-width such a controlled system offers efficient (in most cases optimal) decision procedures for the verification of the controlled system.

We propose a generic approach to define controllable classes of CPDS in terms of quotient graphs, which admit a 'suitable' acyclicity restriction. We also give a generic controller for several of the classes definable in this framework. The controllers we propose are sound and complete for the respective class, meaning that they allow all and only the behaviours of this class. Moreover, our technique for proving the bound on split-width of the controlled systems is also generic and systematic, hence may easily extend to generalisations and other classes as well.

The decidability results for the controllable classes we propose in this thesis are new while they capture, as special cases, several restrictions studied in the literature like bounded phase [LMP07], bounded scope [LN11], poly-forest topology [LMP08a] etc.

We now compare and contrast our results to the related works.

1.2 Comparison with related works

1.2.1 on multi-pushdown systems

Multi-pushdown systems are Turing powerful, and hence even control state reachability problem is undecidable. A main strategy to get around this undecidability is to restrict the behaviours of MPDS in terms of how they access the different stacks. A number of restrictions have been proposed for their under-approximate verification including bounded context switch, bounded phase, ordered multi-pushdown and bounded scope.

Qadeer and Rehof [QR05] showed that bounding the number of switches from using one stack to another (i.e., the number of switches between threads) yields decidability. This restriction, called bounded context switching, has been well-studied [LTKR08, LMP08a, LR09]. The reachability problem of multi-pushdown systems with a bound on number of context switches as a parameter is in NP.

In [LN11] bounded context-switching restriction is extended to bounded-scope where the number of context-switches between a push and the corresponding pop is bounded. In this case, the reachability problem is in PSpace.

An orthogonal generalisation of bounded context switching is the bounded-phase restriction [LMP07] — where a limit is placed on the number of phases, — a phase refers to sequence of steps in which values are popped only from one designated stack while pushing is permitted on all of the stacks. The reachability problem of multi-pushdown systems with a bound on the number of phases as a parameter is in 2ExpTime.

A different restriction (ordered multi-pushdown) [BCCCR96, ABH08] imposes a priority order on the stacks and permits popping only from the non-empty stack with highest priority. The reachability problem of multi-pushdown systems with this restriction is again in 2ExpTime. In [AKS13], a stricter restriction is imposed, where, in addition, pushes are permitted only to the adjacent stacks in the priority ordering. With this restriction, reachability is in ExpTime.

All the above restrictions yield decidability for the control state reachability problem (or emptiness checking). However, the proof of decidability for each class uses ad-hoc techniques.

A unifying proof of decidability is shown via tree-width [MP11, LP12]. In fact [MP11] also captures some classes of message passing concurrent systems. We discuss these in the end of Section 1.2.2.

Using split-width we get another uniform proof of decidability for all the above classes [CGN12a]. While tree-width is defined for general graphs, the definition of split-width is designed for behaviours of CPDS. It is based on simple operations of shuffle and merge and hence, we believe, is easier to handle. A detailed comparison between split-width and tree-width is given in Section 6.1 (Chapter 6).

The proofs of the split-width of the above classes suggested that several of these classes can be jointly generalised preserving decidability.

One such generalisation extends ordered and scope bounded multi-pushdown systems. We freely allow pops of both kinds in this restriction. There is no restriction on pushes. But the corresponding pop a) has to be within fixed number of context switches from then (analogous to time-out) or b) if a) fails, then all such pop events will be ordered on a priority basis (assuming a total order on the priorities of different stacks). Another generalisation is to replace the ordering policy by a bounded phase policy.

These two general classes are shown to be decidable[1]. The decidability is again by bounding split-width, which can be obtained from the bounds of the constituent classes.

Temporal logics and model checking for the above restrictions have been studied in the literature [BCGZ11, ABKS12, LN12, BKM13], but again in an ad-hoc manner.

A generic framework for defining temporal logics for concurrent recursive program is given in [BCGZ11], but a bounded phase is assumed for decidability. [BKM13] refines complexity of such a temporal logic further when the bound on phase is part of the input.

In [LN12] the authors study model checking of multi-pushdown system under scope-bounded and ordering restrictions against temporal logics. The temporal logics used there is expressible in our generic framework of [BCGZ11]. Linear-time model checking under bounded scope restriction is also considered in [ABKS12].

Again, the thesis provides a uniform decision procedure for the satisfiability and model checking of temporal logics for the above restrictions, as well as several other generalisations. It also gives a uniform decision procedure for PDL and MSO satisfiability and model checking problems. The complexities of these uniform decision procedures are optimal in most cases.

When viewing multi-pushdown systems as abstractions of concurrent recursive programs whose actions are ordered by an external scheduler, each of the

[1]A joint generalisation of ordered and phase bounded yields undecidability.

above restrictions can be seen as a particular policy that the external scheduler adopts. Thus under-approximate verification wrt. the above restrictions is a kind of fair model checking, in which those runs which comply to the scheduling policy are verified against some specification.

1.2.2 on message passing concurrent systems

Concurrent systems communicating via unbounded queues are also Turing powerful. Hence the control state reachability problem for even the simple case of finite state systems communicating through unbounded queues (CFMs) is undecidable [BZ83].

In an influential paper, Abdulla and Jonsson [AJ96] showed that, if the underlying queues are lossy then the state reachability problem becomes decidable. For a survey of the plethora of results in the study of lossy channel systems see [Abd10, FS01].

Independently, in the study of message sequence charts (MSCs) [IT11], which model behaviours of communicating finite state machines, a number of decidability results are known. For a survey, see [GKM07, Nar12]. These essentially assume the behaviours to be existentially bounded[2].

Study of communicating recursive systems (CRS) is more recent, following developments in the analysis of multi-threaded programs. In [LMP08a], it is shown that reachability is decidable if the architecture forms a suitable acyclic topology. In [HLMS10], the authors characterise the topologies that give a decidable control state reachability problem under the restriction that local stacks are empty while sending (or instead, receiving) messages.

In [MP11], the authors provide a first unifying reason for why these restrictions work, relating it to the *tree-width* of the graph underlying these runs. If a class of graphs is MSO definable and in addition has bounded tree-width then any MSO expressible property is decidable over such a class [See91, Cou97, Kre09].

In [MP11] it is shown that for bounded context, bounded phase and ordered multi-pushdown restrictions of multi-pushdown systems, as well as for the acyclic topology restrictions [LMP08a] on message passing programs, the class of permissible behaviours when considered as graphs is MSO definable and has bounded tree-width. In [LP12] bounded scope restriction is also shown to have bounded tree-width and MSO definable. As a consequence they get a uniform proof of the decidability of control state reachability for all these classes via tree-width.

In the thesis we introduce new decidable restrictions. These restrictions assume *unbounded* and *reliable* channels and permit behaviours from *arbitrary topologies* which, in particular, are not covered in [LMP08a] and [HLMS10]. Moreover they do not require the local stacks to be empty while accessing queues. The decidability of the new restrictions are again shown via split-width. This captures several of the known decidability results in a uniform framework.

[2]An existentially bounded run admits a linearisation without exceeding the assumed bound on the channels.

Temporal logics and propositional dynamic logics for MSCs have been considered in [Men13b, BKM10, MR04, Pel00]. These works assume the MSCs to be existentially bounded in order to obtain decidability. Moreover, these cannot handle the behaviours of a stack.

The thesis provides uniform decision procedure for the satisfiability and model checking of temporal logics, propositional dynamic logics and monadic second order logics for our new decidable classes. Recall that our restrictions go well beyond existentially bounded MSCs.

1.3 Organisation of the thesis

Formalisms for concurrent processes with data-structures, their behaviours, and specifying properties of behaviours are presented in Chapter 2. We state the various verification problems, and show that they are undecidable.

In Chapter 3 we introduce and illustrate the notion of split-width. The various decision procedures for verification problems with split-width as a parameter are studied in Chapter 4. We also discuss how to adapt the decision procedures for other classes of systems which admit a bound on split-width. The complexities of the decision procedures for such classes are summarised at the end of this chapter. This may be used as a template to deduce the complexity of various classes of CPDS with bounded split-width.

In Chapter 5 split-width is compared and shown to be equivalent to graph parameters such as tree-width and clique-width. Chapter 6 is a discussion, where we give an in-depth comparison between split-width and tree-width and also list out some open problems.

Part III studies various generic classes in a uniform framework, providing decision procedures for various verification problems. These classes generalise various classes studied in the literature. Furthermore a sound and complete distributed controller is provided for each of these classes.

Chapter 7 discusses the desirable features of a controllable class and a controller. Chapter 8 introduces the basic notions necessary to define our classes: how to define a quotient graph of a behaviour based on suitable notions of contexts, and the various notions of acyclicity.

Chapter 9-11 study classes based on acyclic quotient graphs. Chapter 9 presents a class that allows behaviours from arbitrary architectures which are not existentially bounded, thus extending the decidability frontier. The class studied in Chapter 10 and Chapter 11 subsume bounded context switching and bounded phase restrictions of multi-pushdown systems.

All these classes admit distributed controllers that are similar in nature. Hence we present a generic controller for classes admitting finite quotient graphs. The controller for individual classes are obtained by particular instantiation. This generic controller is given in Chapter 9. This generic controller is also proved sound and complete.

Chapter 12 studies another class of restriction which allows unbounded quotient graphs. This class captures bounded scope restriction on multi-pushdown systems. We provide a sound and complete controller for this class based on context-stamping. We conclude Part III in Chapter 13 where we discuss other results and the impacts of our results.

The manuscript concludes in Chapter 14 with further perspectives and directions for future research.

We do not dedicate a chapter on preliminaries in the beginning. Rather, the relevant preliminary notions and results are recalled as and when needed. The reader may benefit from the index given at the very end of this manuscript to easily trace back the definition of some technical terms if needed.

1.4 Other contributions of the author

During my doctoral studies, I have worked on other problems related to verification as well. While the results of this thesis are obtained in collaboration with Paul Gastin and K. Narayan Kumar, the following works have been done in collaboration with Benedikt Bollig, Loïc Hélouët, Paul gastin, Ahmet Kara, K. Narayan Kumar, Thomas Schwentick and Marc Zeitoun.

1.4.1 Temporal logics for concurrent recursive programs communicating via shared variables

We study the problem of modelling, specifying and model checking concurrent recursive programs communicating via shared variables. Multi-pushdown systems, while being a nice model to study their interleaving semantics, are not very satisfactory to model the intrinsic concurrency. Hence we need a formalism which models both concurrency and recursion. Further we need specification formalisms which can reason about these two indispensable aspects. Finally we need efficient algorithms for model checking.

For sequential non-recursive systems, linear-time temporal logic (LTL) [Pnu77] is a yardstick among the specification languages. It combines high expressiveness (equivalence to first-order logic [Kam68]) with a reasonable complexity of decision problems such as satisfiability and model checking. As real programs are often concurrent or rely on recursive procedures, LTL has been extended in two directions

First, asynchronous finite-state programs (asynchronous automata) [Zie87] are a formal model of shared-memory systems and properly generalise finite-state sequential programs. Their executions are no longer sequential (i.e., totally ordered) but can be naturally modelled as graphs or partial orders. In the literature, these structures are known as *Mazurkiewicz traces.* They look back on a long list of now classic results that smoothly extend the purely sequential setting (e.g., expressive equivalence to first-order logic) [DR95, DG06].

Second, in an influential paper, Alur and Madhusudan extend the finite-state sequential model to *visibly pushdown automata* (VPA) [AM09]. VPA are a flexible model for recursive programs, where subroutines can be called and executed while the current thread is suspended. The execution of a VPA is still totally ordered. However, it comes with some extra information that relates a subroutine call with the corresponding return position, which gives rise to the notion of *nested words* [AM09]. Alur et al. recently defined versions of LTL towards this infinite-state setting [AEM04, AAB$^+$08] that can be considered as canonical counterparts of the classical logic introduced by Pnueli.

To model programs that involve both recursion and concurrency, one needs to mix both views. A first model for concurrent recursive programs with partial-order semantics was considered in [BGH09]. Executions of their *concurrent VPA* equip Mazurkiewicz traces with multiple nesting relations. For decidability, an existential bound on the number of phases [LMP07] is assumed. This serves as a good behavioural model, which models recursion and at the same time preserves independencies between program events.

In [BCGZ11], we present linear-time temporal logics for concurrent recursive programs. A temporal logic is parametrized by a finite set of modalities that are definable in monadic second-order logic (cf. [GK03]). In addition, it provides path expressions similar to those from PDL [FL79] or XPath [Lib06], which are orthogonal to the modalities. This general framework captures temporal logics considered in [AEM04, AAB$^+$08, DK11] when we restrict to one process, and it captures those considered in [DG06, GK03, GK10] when we go without recursion. Our decision procedures for the (bounded phase) satisfiability problem are optimal in all these special cases, but provide a unifying proof. They also apply to other structures such as ranked and unranked trees. We then use our logics for model checking. To do so, we provide a system model similar to [BGH09], which we call concurrent recursive Kripke systems, whose behavioural semantics preserves concurrency (unlike multi-pushdown systems). The complexity upper bounds from satisfiability are preserved. More concretely, satisfiability and model checking are decidable in ExpTime and 2ExpTime, depending on the precise path modalities. We also show matching lower bounds.

Summarizing, we provide the first framework to specify linear-time properties of concurrent recursive programs appropriately over partial orders.

1.4.2 Systems handling data

So far we have considered boolean programs. Data, potentially unbounded, is a main feature of many real programs. However, data is quite difficult to handle as almost all decidable verification problems become undecidable as soon as an infinite data domain is assumed. The behaviours are called data-words, which are words adorned with a data-value from an infinite domain at every position. The undecidability holds even for finite state sequential programs without recursion. This is the case even when the data is used in a restrictive manner, say only for equality checks (and no arithmetic). Even limiting the specification language to first order logic instead of MSO renders satisfiability undecidable as soon as a data equality predicate is allowed.

A wide range of automata and grammars over data words have been introduced in the literature [KF94, Tze11, KZ10, GKS10, BDM$^+$11, CK98, Bol11]. For all of them, MSO model checking is undecidable. Model checking of counter machines against freeze LTL was considered in [DLS08, DS10]. However the temporal logic, which can be embedded into MSO logic, has to be restricted further to obtain decidability results.

Our contribution [BCGK12] The undecidability is probably because uncontrolled occurrences of the data values in a word makes it difficult to track equal values. However, for many practical applications, the repetition of a same data value is often not by chance, but on purpose[3]. For example in protocol descriptions in a dynamic distributed system, a process identity (pid) may be stored in a register for a later reuse. Storing a data value in a data structure and reusing it later may allow us to track it. We explore this idea in [BCGK12], considering programs which use data-structures to store data. For this we consider a model which is both an extension and a restriction of register automata [KF94]. We extend the register automata with several stacks, increasing it modelling power. At the same time we forbid arbitrary repetitions of data value, thus gaining decidability for MSO model checking. We call this model a data multi-pushdown automata.

A data-pushdown automaton has a stack and a set of registers to facilitate storage of data-values. Moreover, transitions can be guarded by data comparison tests on the current register contents and the top of the stack. The generated data word may use values stored in the registers, or "fresh" values which are different from any value ever used in the history. Model-checking of data-pushdown automata against MSO with data equality tests is decidable. A data-multi-pushdown automaton extends a data-pushdown automaton by allowing multiple stacks. We can recover decidability of MSO model-checking with any restriction on multi-pushdown systems which yields MSO decidability (like scheduling policies or bounded split-width).

A data multi-pushdown automaton is powerful enough to model protocols like peer-to-peer protocol and leader election protocol in a distributed setting with dynamic process creation. It can also handle broadcasting. For such applications, the role of the stack is rather to aid the storage and retrieval of the pids than modelling recursion.

In [Cyr12] we observe that the above results can be extended to data-domain with an ordering. The MSO logic, is also extended to include data-comparisons. The freshness assumption requires the fresh data-value to be higher than any previously used value. Note that, this is in accordance with data-values originating from time-stamping or with the UNIX pid-assigning conventions.

[3]The probability that a same data-value repeats by chance is zero, since the data-domain is infinite.

1.4.3 Dynamic distributed systems

We apply our decidability results on systems handling data to the special case of dynamic distributed systems, which are interesting especially in the setting of this thesis. Dynamic distributed systems are concurrent systems with dynamic process creation. Many concurrent systems we encounter have the ability to spawn processes, thus there is no bound on the maximum number of processes the whole system can have. The internet is a typical example. Thus these systems handle an unbounded data (their process identifiers, or pids). Such systems are usually physically distributed and hence employ message passing as a feasible communication mechanism instead of shared memory. We propose a formalism to model protocols for dynamic distributed systems.

Branching high-level message sequence charts [BCH$^+$13] In scenarios with only fixed number of processes and no dynamic process creation, high-level message sequence charts have been employed successfully for protocol description. We extend this formalism to incorporate dynamic process creation as well. The extended formalism, called a branching high-level message sequence chart, is an automaton with several registers and an implicit stack (or, it can branch off like a tree, and the branches can join later). This allows to describe some protocols like peer-to-peer protocol in a dynamic setting. A branching high-level message sequence chart may be seen as a global automaton which describes the global (intended) behaviour of the system.

Branching high-level message sequence charts can be encoded as a data push-down automaton. MSO over message sequence charts can be translated to MSO over data words. Thus MSO model checking of branching high level message sequence charts is again decidable.

Branching high-level message sequence charts are quite convenient to describe a protocol, but they are not models of real distributed implementation. An implementation model for dynamic distributed systems is described next.

Dynamic communicating automata [BCH$^+$13] Dynamic communicating automata are finite state machines with several registers. At any instant of time, there are finitely many processes, each with a unique process identifier (pid). Each of these processes is executing a copy of the finite state machine with registers. The registers store pids of some other processes in the network. Since there are only a fixed number of registers, a process can remember only a bounded number of pids at a time. Processes can 1) send messages to any process it remembers (in other words, whose pid is stored in its register), 2) receive messages from other processes and 3) create a new process with a "fresh" pid. The messages in addition may contain other pids. The receiving process may choose to store some of the message contents (pids) in some of its registers. Hence the communication neighbourhood of a process changes dynamically as the system evolves. The communication is via FIFO channels. We assume an unbounded FIFO channel between every pair of processes in each direction.

This powerful natural model however renders basic problems like control state reachability undecidable. It is so, even if we restrict the channel capacity to be zero (that is, only synchronous communication is allowed) and bound the number of registers to two. However, from some formal high level protocol description, we may *synthesise* an implementation, the implementation being correct by construction. We employ branching high level message sequence charts for protocol description as they can be verified against MSO. Synthesis of a dynamic communicating automaton from a branching high-level message sequence chart is not always possible. However, we identify syntactic subclasses of branching high-level message sequence charts for which synthesis is always possible, and provide an efficient algorithm for synthesis in this case.

To summarise, branching high-level message sequence charts act as a low-level specification formalism which can be model checked against MSO. Once the low-level specification formalism is guaranteed to specify the intended requirements, we implement it directly as a dynamic communicating automaton, which is now formally verified by virtue of the construction.

Related Work The extension of high-level message sequence charts to branching high-level message sequence charts is inspired by branching automata [LW00, LW01] and register automata [KF94]. Several other formalisms with dynamic process creation can be found, for example, in [LMM02, BGP08, BS01, Mey08, BLP08, ABQ11]. Dynamic communicating automata were introduced in [BH10]. However, their version could not handle pids as message contents. Also, we optimally solve the implementability problem for a class of specifications that cannot be handled by [BH10].

Selected publications by the author

[BCGK12] Benedikt Bollig, Aiswarya Cyriac, Paul Gastin, and K. Narayan Kumar. Model checking languages of data words. In *FoSSaCS*, volume 7213 of *Lecture Notes in Computer Science*, pages 391–405. Springer, 2012.

[BCGZ11] Benedikt. Bollig, Aiswarya. Cyriac, Paul. Gastin, and Marc. Zeitoun. Temporal logics for concurrent recursive programs: Satisfiability and model checking. In *MFCS*, volume 6907 of *Lecture Notes in Computer Science*, pages 132–144. Springer, 2011.

[BCH$^+$13] Benedikt Bollig, Aiswarya Cyriac, Loïc Hélouët, Ahmet Kara, and Thomas Schwentick. Dynamic communicating automata and branching high-level MSCs. In *LATA*, volume 7810 of *Lecture Notes in Computer Science*, pages 177–189, Bilbao, Spain, April 2013. Springer.

[CGN12a] Aiswarya Cyriac, Paul Gastin, and K. Narayan Kumar. MSO decidability of multi-pushdown systems via split-width. In *CONCUR*, volume 7454 of *Lecture Notes in Computer Science*, pages 547–561. Springer, 2012.

[Cyr12] Aiswarya Cyriac. Model Checking Dynamic Distributed Systems. In *Winter School Modelling and Verifying Parallel processes (MOVEP'12)*, Marseille, France, December 2012.

Chapter 2

Systems of concurrent processes with data-structures

Contents

In this chapter we will consider systems with fixed number of finite state processes with access to fixed number of unbounded stack and queue data-structures. This forms an interesting class of systems on its own right and at the same time captures many important classes of systems in a uniform way.

2.1 Introduction

The study of communicating concurrent systems is an important and challenging research area. Various means of interaction between the concurrent processes have been considered by researchers, of which two prominent ones are 'shared-variable communication' and 'communication via channels'.

Different processes running on the same machine may employ 'shared-variable communication' as the main method of interaction, while physically distributed programs rely on first-in-first-out communication channels. The former corresponds to 'synchronous communication' while the latter gives rise to 'asynchronous communication'.

Recursive programs communicating via shared variables are also called multi-threaded programs. Such systems can be modelled as a multi-pushdown systems for studying their interleaving semantics. The concurrency is more visible in the models like concurrent recursive Kripke systems [BGH09] (whose behaviours are called nested-traces). Concurrent programs communicating via channels look back at the rich theory of message sequence charts [IT11].

We will consider generic systems which incorporate both these methods of communication. We will be considering physically distributed machines which communicate via (possibly several) reliable first-in-first-out queues. Each of these machines are capable of running multi-threaded programs. These programs within a machine use shared variable for communication. Moreover, these programs may use stacks and queues as data-structures to aid their local computation.

This models real-word systems like several computers connected via a local-area-network. Each of these computers may run several recursive programs concurrently, which may communicate among themselves via shared memory access. A computer may also provide several data-structures to the programs for the storage and retrieval of data. These computers may access the network to communicate to its peers, where the message are sent and received via first-in-first-out channels.

2.2 Architecture

A *system of concurrent processes with data-structures (CPDS)* has a fixed finite set of $\mathfrak{p}$ many finite state processes denoted by $\mathbf{Procs} = \{1, \ldots, \mathfrak{p}\}$.

Moreover the system has a finite set of data-structures denoted $\mathbf{DS}$. The number of data-structures is denoted $\mathfrak{d}$, i.e, $\mathfrak{d} = |\mathbf{DS}|$. The data-structures are either stacks or queues. Thus $\mathbf{DS} = \mathbf{Stacks} \uplus \mathbf{Queues}$. We denote by $\mathfrak{s}$ (resp. $\mathfrak{q}$) the number of stack (resp. queue) data-structures: $\mathfrak{s} = |\mathbf{Stacks}|$ and $\mathfrak{q} = |\mathbf{Queues}|$.

We also keep the set of processes and the set of data-structures disjoint. $\mathbf{Procs} \cap \mathbf{DS} = \emptyset$.

Stacks and queues may be used as local data-structures by processes. In addition queues may also serve as FIFO communication channels between processes. Hence each stack data-structure can be accessed by only one process. A queue data-structure can be written on to by one process and read from also by one process. Note that, the **read** operations on these data-structures **are destructive**: the data item is removed from the data-structure once it is read. Thus write operations as well as read operations "modify" the data-structure.

The read and write accesses are specified by mappings $\mathsf{Writer} : \mathbf{DS} \to \mathbf{Procs}$ and $\mathsf{Reader} : \mathbf{DS} \to \mathbf{Procs}$. We also define the symmetric mappings $\mathsf{W\text{-}access} : \mathbf{Procs} \to 2^{\mathbf{DS}}$ and $\mathsf{R\text{-}access} : \mathbf{Procs} \to 2^{\mathbf{DS}}$ which identifies the subset of data-structures to which a process has write-access and read-access respectively. For all $p \in \mathbf{Procs}$ and all $d \in \mathbf{DS}$, $\mathsf{Writer}(d) = p$ if and only if $d \in \mathsf{W\text{-}access}(p)$ and $\mathsf{Reader}(d) = p$ if and only if $d \in \mathsf{R\text{-}access}(p)$.

Since stacks are local to processes, $\mathsf{Writer}(d) = \mathsf{Reader}(d)$ for all $d \in \mathbf{Stacks}$. Thus the stacks can be used to aid the local computation and the computing power, like permitting recursive calls. The queues in addition can act as unbounded FIFO communication channels between the processes. Notice that a process may be equipped with several stacks and queues, and there may be several FIFO channels between a pair of processes.

We call the above settings of a system of CPDS an architecture. Thus an *architecture* is a tuple

$$\mathfrak{A} = (\mathbf{Procs}, \mathbf{Stacks}, \mathbf{Queues}, \mathsf{Writer}, \mathsf{Reader})$$

where these objects are defined as above.

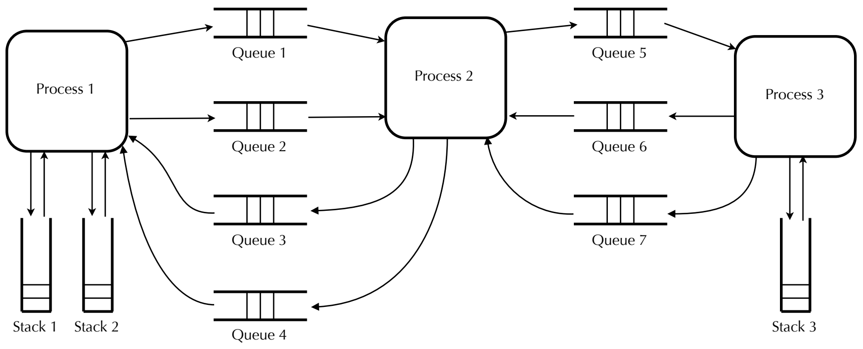


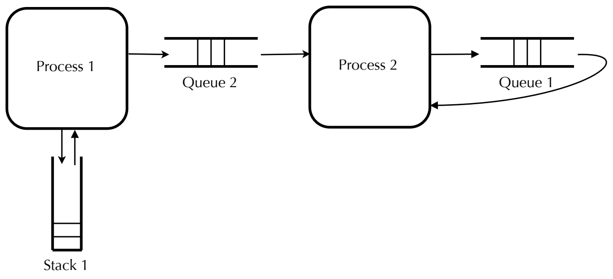
Figure 2.1: Architecture $\mathfrak{A}_1$

Example 2.1. In Figure 2.1 we depict an architecture with three processes, three stacks and seven queues. All queues serve as FIFO communication channels. Process 1 has two stacks to aid its local computations, Process 3 has one, but Process 2 does not have any local stacks. There are two channels in each direction between Process 1 and Process 2. There are two channels from Process 3 to Process 2 as well, but only one channel from Process 2 to Process 3. Process 1 and Process 3 cannot communicate directly, as there are no channels between them. This architecture could be described as in Table 2.1.

Procs	=	$\{1,2,3\}$
Stacks	=	$\{s_1, s_2, s_3\}$
Queues	=	$\{q_1, q_2, q_3, q_4, q_5, q_6, q_7\}$
Writer	=	$\{s_1 \mapsto 1, s_2 \mapsto 1, s_3 \mapsto 3, q_1 \mapsto 1, q_2 \mapsto 1, q_3 \mapsto 2, q_4 \mapsto 2, q_5 \mapsto 2, q_6 \mapsto 3, q_7 \mapsto 3\}$
Reader	=	$\{s_1 \mapsto 1, s_2 \mapsto 1, s_3 \mapsto 3, q_1 \mapsto 2, q_2 \mapsto 2, q_3 \mapsto 1, q_4 \mapsto 1, q_5 \mapsto 3, q_6 \mapsto 2, q_7 \mapsto 2\}$

Table 2.1: Architecture $\mathfrak{A}_1$

Example 2.2. Another architecture with two processes, a stack, and two queues is described in Table 2.2. Here one queue acts as a communication channel, and the other serves as a local data-structure.



Architecture $\mathfrak{A}_2$

Procs	=	$\{1,2\}$
Stacks	=	$\{s_1\}$
Queues	=	$\{q_1, q_2\}$
Writer	=	$\{s_1 \mapsto 1, q_1 \mapsto 2, q_2 \mapsto 1\}$
Reader	=	$\{s_1 \mapsto 1, q_1 \mapsto 2, q_2 \mapsto 2\}$

Table 2.2: Architecture $\mathfrak{A}_2$

What other data-structures? We are considering only stack and queue data-structures in this thesis. However, it is possible to consider architectures with other types of data-structures such as bags and linked lists. One may also consider architectures in which a data-structure may have several writers and readers. We believe it is possible to extend the results of this thesis into this more general framework.

Another setting may permit non-destructive reads on a data-structure by a subset of processes. However, these raise 'race'-issues, which need to be carefully handled at the semantic and specification level. This is not in the scope of this thesis.

2.3 Modelling the system

To define a system of concurrent processes with data-structures we first fix an architecture $\mathfrak{A}$. Each process in $\mathfrak{A}$ may perform actions from a finite set of actions Σ. Thus a model of the system will be parametrised by the architecture and the set of actions.

2.3.1 Concurrent Processes with Data-Structures

Definition 2.3. A system of concurrent processes with data-structures (abbreviated as CPDS) over $\mathfrak{A}$ and Σ is a tuple $\mathcal{S} = (\mathsf{Locs}, \text{Trans}, (\ell_1^{\text{in}}, \ldots, \ell_{\mathfrak{p}}^{\text{in}}), \mathsf{Locs}_{\text{fin}})$ where `CPDS`

- Locs is the finite set of control locations.
- $(\ell_1^{\text{in}}, \ldots, \ell_{\mathfrak{p}}^{\text{in}}) \in \mathsf{Locs}^{\mathbf{Procs}}$ is global initial state, which lists out the local initial control location for each process.
- $\mathsf{Locs}_{\text{fin}} \subseteq \mathsf{Locs}^{\mathbf{Procs}}$ is the set of global final states.
- Trans is a tuple $(\text{Trans}_p)_{p \in \mathbf{Procs}}$, where Trans_p is the set of local transitions of Process p.

The local transitions of a process p is partitioned into the following:

$$\begin{aligned}\text{Trans}_p = &\text{Trans}_{p:int} \\ &\uplus \biguplus_{d \in \mathsf{W\text{-}access}(p)} \text{Trans}_{p \to d} \\ &\uplus \biguplus_{d \in \mathsf{R\text{-}access}(p)} \text{Trans}_{p \leftarrow d}\end{aligned}$$

where $\text{Trans}_{p:int}$ denotes the internal (no data-structure access) transitions of Process p, $\text{Trans}_{p \to d}$ are the transitions of Process p which write to data-structure d, $\text{Trans}_{p \leftarrow d}$ are the transitions of Process p which read from data-structure d, Thus:

$$\begin{aligned}\text{Trans}_{p:int} &\subseteq \mathsf{Locs} \times \Sigma \times \mathsf{Locs} \\ \text{Trans}_{p \to d} &\subseteq \mathsf{Locs} \times \Sigma \times \mathsf{Locs} \times \mathsf{Locs} \\ \text{Trans}_{p \leftarrow d} &\subseteq \mathsf{Locs} \times \mathsf{Locs} \times \Sigma \times \mathsf{Locs}\end{aligned}$$

For an immediate and intuitive understanding of CPDS, we will now give an operational semantics. This describes the sequential (or interleaving) evolution of the CPDS and is not really concurrent in nature. In the next section we will see the more interesting partial-order semantics of a CPDS.

Intuitively, a sequential execution (or a 'linearisation' of a run) of a CPDS starts in an initial configuration, and it changes the configurations as permitted by the transition relations, and in order to be successful, it ends in a final accepting configuration.

The set of configurations of the CPDS $\mathcal{S}$ is $C_{\mathcal{S}} = \mathsf{Locs}^{\mathbf{Procs}} \times (\mathsf{Locs}^*)^{\mathbf{DS}}$. A configuration $c = ((\ell_p)_{p \in \mathbf{Procs}}, (u_d)_{d \in \mathbf{DS}})$ indicates that the current local state of process p is ℓ_p and the current contents of data-structure d is u_d.

The configuration c is *initial* if $(\ell_1, \ldots \ell_{\mathtt{p}}) = (\ell_1^{\text{in}}, \ldots, \ell_{\mathtt{p}}^{\text{in}})$ and $u_d = \epsilon$ for all $d \in \mathbf{DS}$. Similarly the configuration c is *final* if $(\ell_1, \ldots \ell_{\mathtt{p}}) \in \mathsf{Locs}_{\text{fin}}$ and $u_d = \epsilon$ for all $d \in \mathbf{DS}$.

We define the *evolves relation* which describe one step evolution of the configuration. It is actually a union of $\xrightarrow{a}_p \subseteq C_{\mathcal{S}} \times C_{\mathcal{S}}$ where $a \in \Sigma$ and $p \in \mathbf{Procs}$. The relation $\xrightarrow{a}_p$ is defined as follows where $\ell_p, \ell'_p, \ell \in \mathsf{Locs}$

$$
\begin{aligned}
(\ldots, \ell_p, \ldots) &\xrightarrow{a}_p (\ldots, \ell'_p, \ldots) && \text{if } (\ell_p, a, \ell'_p) \in \text{Trans}_{p:int} \\
(\ldots, \ell_p, \ldots, u_d, \ldots) &\xrightarrow{a}_p (\ldots, \ell'_p, \ldots, u_d\ell, \ldots) && \text{if } (\ell_p, a, \ell'_p, \ell) \in \text{Trans}_{p \to d} \\
(\ldots, \ell_p, \ldots, u_d\ell, \ldots) &\xrightarrow{a}_p (\ldots, \ell'_p, \ldots, u_d, \ldots) && \text{if } (\ell_p, \ell, a, \ell'_p) \in \text{Trans}_{p \leftarrow d} \\
& && \quad \text{and } d \in \mathbf{Stacks} \\
(\ldots, \ell_p, \ldots, \ell u_d, \ldots) &\xrightarrow{a}_p (\ldots, \ell'_p, \ldots, u_d, \ldots) && \text{if } (\ell_p, \ell, a, \ell'_p) \in \text{Trans}_{p \leftarrow d} \\
& && \quad \text{and } d \in \mathbf{Queues}
\end{aligned}
$$

`seq-run` A *sequential run* ρ of $\mathcal{S}$ is a sequence $c_0 \xrightarrow{a_1}_{p_1} c_1 \xrightarrow{a_2}_{p_2} c_2 \ldots \xrightarrow{a_n}_{p_n} c_n$ of configurations and actions, for $n \geq 1$ and c_0 initial. It is *accepting* if c_n is final. Sequential run is abbreviated as seq-run for the rest of this thesis.

Remark 2.4. Several processes of a CPDS may progress concurrently (or independently). This information is not conveyed by a sequential run. The independency will be explicitly reflected in the partial-order semantics of a CPDS (cf. Section 2.4.2).

`Data-Structure alphabet` **No separate data-structure alphabet!** Notice that we do not have employ a separate data-structure alphabet. Very often in formal language theory, automata formalisms with auxiliary storage employ a separate finite alphabet for each auxiliary data-structure. However, we have decided not to follow this convention.

By not insisting on a fixed data-structure alphabet, we do not limit the contents of the data-structure and their possible functionalities. In fact our models let any local state to be written on to the data-structures by the local processes. Thus the data-structure alphabet of a system is as liberal as the set of local states. This is clearly more generic than keeping separate data-structure alphabet, as one could easily augment the finite set of states to include a finite data-structure alphabet.

We believe it allows us to easily model programs and protocols described in high-level languages in our setting. As we mentioned before, one purpose of the stacks is to model recursion. Also, one purpose of queues is to serve as FIFO communication channels. We do not want to restrict what can be written onto these data-structures in the architecture level.

If we consider high level programming languages supporting recursion, they do not use or define a separate stack-alphabet. In fact, the program stack can store the program counter value and the local 'state' which gives the assignments to each local variables. Hence on returning from a recursive call, the program is able to restore the local state and to continue the computation. However, if the program stack was bound to use only a fixed finite alphabet, it limits the possibilities of recursion.

One may argue, rightly so, that the actual unbounded values can be encoded in the finite alphabet. But this alters the behaviour – increasing the number of stack-accesses by a logarithmic factor of the size of the program. Also, one would want to specify requirements about a program without actually looking at the size of a program. The requirements, when formalised, are anticipated to be independent of the evaluation model. This is mainly because one would like to translate the requirements specified in one formalism to another, independently.

Similarly, for the case of communication channel via queues, the amount of information that can be communicated between processes is bounded if we insist on finite alphabet. As before, if we try to encode more information via a finite alphabet, this will change the underlying behaviour of the system: More messages need to be sent in order to transmit the information in one message.

Changing the underlying behaviours may change the decidability status of various problems on these models. For example, consider a class of distributed systems which comply to a certain protocol. This protocol may have nice properties like the channel size never exceeds a pre-determined bound during the execution. Many algorithms are available for various verification problems on systems with such nice properties. However, if we encode the desired information by an a priori fixed finite alphabet, the change in the underlying behaviour may make them violate the nice property. Thus the known verification methods which rely on the nice properties cannot be used anymore for the verification of these systems.

One main topic of study in formalisms with fixed data-structure alphabet is the data-structure language. To study data-structure languages in our formalisms, we may take homomorphic projections from the control states to the desired finite alphabet. This aspect is discussed again in forthcoming remarks.

Alternative Another equivalent definition of CPDS may assume that always the current local state is written into data-structure. This simplifies the write-transitions to have the form $\text{Trans}_{p \to d} \subseteq \mathsf{Locs} \times \Sigma \times \mathsf{Locs}$.

However this definition makes the local state of a process transparent to its communication partners. On the other hand, our current definition allows a better privacy policy for the processes. The local states are protected. The processes need to communicate only the information they want to.

Behaviours of the system An accepting seq-run, which is a sequence of configurations and actions, gives the complete details about one possible behaviour of the system.

However, this complete and natural representation of behaviours has certain drawbacks:

- An accepting seq-run is a word over an infinite alphabet (the data-structures are unbounded, and hence the configurations are infinite).
- Since the seq-runs are linear, the concurrency information is lost.
- The related accesses on data-structure is not explicitly linked.

We propose another representation of the behaviours of the system which will avoid the above shortcomings. These are essentially graphs, whose nodes represent the events of the CPDS. Each node is labelled by the action label, the process which executes it, and the data-structure it accesses, if any. There are process-successor edges which link successive events on the same process. Thus two independent events on two processes are not necessarily ordered, which makes it a succinct representation for a “set of linearizations”, and the concurrency information is preserved. Also, the matching write and read events are linked by a direct edge in the graph, making the data-structure accesses explicit.

Making the matching relations of data-structures explicit allows us to specify properties of the systems pertaining to the data-structure accesses, and subsequently, a data-structure aware verification of these systems.

We will see this representation next, and we show that we do not lose any information about the behaviours by choosing this representation. Thus the language of a CPDS will be defined as a set of such graphs.

2.4 Modelling the behaviour

In order to represent the behaviours of a system of concurrent processes with data-structures, we extend Message Sequence Charts with nesting relations. Message sequence charts are scenarios standardised by ITU [IT11]. In their simplest form, they describe finite message exchanges among a finite set of processes, or the queue behaviour. Similarly nested words [AM09] enrich a word with nesting relations in order to describe behaviours of a recursive system, or the stack behaviour. Since we need to describe both queue and stack behaviours, we integrate MSCs and nested words to yield Message Sequence Charts with Nesting (MSCN).

Message sequence charts with nestings are essentially sequences of events adorned with both message relations and nesting relations. We will define them formally now.

2.4.1 Message sequence charts with nestings

A message sequence chart with nesting (MSCN) over an architecture $\mathfrak{A}$ consists of a number of events E executed by the processes in $\mathfrak{A}$.

Each event is labelled by its action (λ), the process which executes it (pid), and the data-structure it accesses (δ), if any.

The set of events executed by Process p is totally ordered by a direct-successor relation $\rightarrow$. The relation $\rhd$ associates each write event on some data-structure with the corresponding read event.

The exchange of messages via any queue has to conform with a FIFO policy. Similarly, the push and the corresponding pop on any stack has to conform with a LIFO policy.

Definition 2.5 (MSCN)**.** A *message sequence chart with nesting (MSCN)* over an architecture $\mathfrak{A}$ and a set of actions Σ is a tuple MSCN

$$\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \delta, \rightarrow, \rhd)$$

where

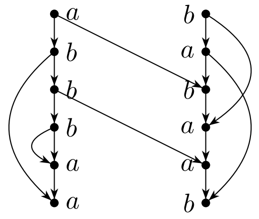
- $\mathcal{E}$ is a non-empty finite set of *events*
- the mapping $\lambda \colon \mathcal{E} \rightarrow \Sigma$ labels each event with an action label,
- the mapping $\mathsf{pid} \colon \mathcal{E} \rightarrow \mathbf{Procs}$ assigns a process to each event,
- the partial mapping $\delta \colon \mathcal{E} \rightarrow \mathbf{DS}$ labels relevant events with a data-structure id,
- the underlying graph $(\mathcal{E}, (\rightarrow \cup \rhd)^*)$ is a partial order,
- $\rightarrow \subseteq \bigcup_{p \in \mathbf{Procs}} \mathsf{pid}^{-1}(p) \times \mathsf{pid}^{-1}(p)$ and, for each $p \in \mathbf{Procs}$, $\rightarrow$ restricted to $\mathsf{pid}^{-1}(p)$ is the direct-successor relation of some total order on $\mathsf{pid}^{-1}(p)$,
- the matching relation is the union of the matching relations of each data-structure: $\rhd = \biguplus_{d \in \mathbf{DS}} \rhd^d$ where $\rhd^d \subseteq \delta^{-1}(d) \times \delta^{-1}(d)$. Moreover,
  - For each $e \in \mathsf{dom}(\delta)$, there is $f \in \mathcal{E}$ such that $e \rhd f$ or $f \rhd e$.
  - If $e \rhd f$, then $\delta(e) = \delta(f) = d$, say, and $\mathsf{Writer}(d) = \mathsf{pid}(e)$ and $\mathsf{Reader}(d) = \mathsf{pid}(f)$.

- $\rhd$ is an irreflexive and vertex-disjoint matching relation on $\mathsf{dom}(\delta)$: If $(e_1, f_1) \in \rhd$ and $(e_2, f_2) \in \rhd$ and if $(e_1, f_1) \neq (e_2, f_2)$, then $|\{e_1, f_1, e_2, f_2\}| = 4$. Moreover, if $(e_1, f_1) \in \rhd$ then $e_1 \neq f_1$.
- For each $d \in$ **Stacks**, $\rhd^d$ is a nesting relation which conforms to LIFO: if $e_1 \rhd^d f_1$ and $e_2 \rhd^d f_2$ are different nesting edges then we do not have $e_1 \to^+ e_2 \to^+ f_1 \to^+ f_2$.
- For each $d \in$ **Queues**, $\rhd^d$ is a message relation which conforms to FIFO: if $e_1 \rhd^d f_1$ and $e_2 \rhd^d f_2$ are different message edges then we do not have $e_1 \to^+ e_2$ and $f_2 \to^+ f_1$.

MSCNs enjoy a natural graphical representation.

Example 2.6. An MSCN over $\mathfrak{A}_1$ (cf. Example 2.1) and a unary alphabet is shown in Figure 2.2. Each process is represented by a vertical line. The relation $\to$ orders (top-down) consecutive events located on the same process line. The messages are depicted by straight edges (solid or dotted) connecting different lines and nesting edges are depicted by curved edges (solid or dotted) connecting within a line. More specifically, $\rhd^{s_1}$ is depicted by solid curved edges on Process 1, $\rhd^{s_2}$ by dotted curved edges on Process 1 and $\rhd^{s_3}$ is depicted by solid curved edges on Process 3. $\rhd^{q_1}$ is depicted by solid straight edges from Process 1 to Process 2, $\rhd^{q_2}$ by dotted straight edges from Process 1 to Process 2, $\rhd^{q_3}$ by solid straight edges from Process 2 to Process 1, $\rhd^{q_4}$ by dotted straight edges from Process 2 to Process 1, $\rhd^{q_5}$ by solid straight edges from Process 2 to Process 3, $\rhd^{q_6}$ by solid straight edges from Process 3 to Process 2, and $\rhd^{q_7}$ is depicted by dotted straight edges from Process 3 to Process 2.

Example 2.7. An MSCN over Architecture $\mathfrak{A}_2$ and $\Sigma = \{a, b\}$ is shown on the right.



$\mathbb{MSCN}(\mathfrak{A}, \Sigma)$ The set of all MSCNs over Architecture $\mathfrak{A}$ and set of actions Σ is denoted $\mathbb{MSCN}(\mathfrak{A}, \Sigma)$.

Remark 2.8. Notice that the degree of any node of an MSCN is at most three.

2.4.2 Languages of MSCNs

We can view a CPDS as an acceptor for MSCNs. For this we define a notion of a *partially-ordered-run* (abbreviated as po-run) of a CPDS over an MSCN. A po-run labels the events of an MSCN with the control locations of the CPDS. This is given by the control location labelling c-loc : $\mathcal{E} \to \mathsf{Locs}$. In order to keep track of the control locations stored in the data-structures, some nodes need to be labelled by an additional location which is specified by the data-structure location mapping d-loc : $\mathcal{E} \to \mathsf{Locs} \uplus \{\bot\}$. Thus a po-run of a CPDS on an MSCN is a pair of mappings (c-loc, d-loc).

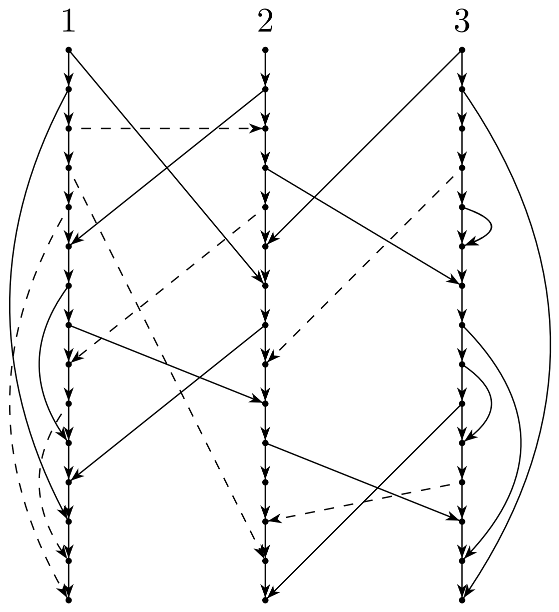


Figure 2.2: An MSCN over Architecture $\mathfrak{A}_1$

A *partially-ordered-run* of a CPDS $\mathcal{S} = (\mathsf{Locs}, \text{Trans}, (\ell_1^{\text{in}}, \ldots, \ell_{\mathsf{p}}^{\text{in}}), \mathsf{Locs}_{\text{fin}})$ over an MSCN $\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \delta, \rightarrow, \rhd)$ is a pair (c-loc, d-loc) such that the following consistency conditions hold for all $e \in \mathcal{E}$. We denote by e^-, the unique event such that $e^- \rightarrow e$ if it exists, and otherwise $e^- = \perp_{\mathsf{pid}(e)} \notin \mathcal{E}$. We set $\text{c-loc}(\perp_p) = \ell_p^{\text{in}}$ as a convention. `po-run`

1. if $e \rhd f$ then $(\text{c-loc}(e^-), \lambda(e), \text{c-loc}(e), \text{d-loc}(e)) \in \text{Trans}_{\mathsf{pid}(e) \rightarrow \delta(e)}$
2. if $f \rhd e$ then $(\text{c-loc}(e^-), \text{d-loc}(f), \lambda(e), \text{c-loc}(e)) \in \text{Trans}_{\mathsf{pid}(e) \leftarrow \delta(e)}$ and $\text{d-loc}(e) = \perp$
3. if e is not part of $\rhd$, then $(\text{c-loc}(e^-), \lambda(e), \text{c-loc}(e)) \in \text{Trans}_{\mathsf{pid}(e):int}$ and $\text{d-loc}(e) = \perp$

Let e_p denote the maximal event on process p if it exists. Otherwise, we set $e_p = \perp_p \notin \mathcal{E}$ and as we set before $\text{c-loc}(\perp_p) = \ell_p^{\text{in}}$ as a convention. A po-run is *accepting* if $(\text{c-loc}(e_1), \ldots, \text{c-loc}(e_{\mathsf{p}})) \in \mathsf{Locs}_{\text{fin}}$.

Thus the *language* accepted by a CPDS $\mathcal{S}$ is the set of MSCNs on which it has an accepting po-run. We denote it by $\mathscr{L}_{\text{po}}(\mathcal{S})$. $\mathscr{L}_{\text{po}}(\mathcal{S})$

We have an understanding of the behaviour of a CPDS in a operational way (cf. seq-runs) from Section 2.3. Here we see the CPDS as an acceptor of MSCNs via po-runs. Now we will see that, po-runs and seq-runs of a CPDS $\mathcal{S}$ are closely related.

We start by associating an MSCN to a sequential run ρ.

Let $\rho = c_0 \xrightarrow{a_1}_{p_1} c_1 \xrightarrow{a_2}_{p_2} c_2 \dots \xrightarrow{a_n}_{p_n} c_n$ be a seq-run of a CPDS $\mathcal{S}$. The MSCN associated to ρ, $\mathcal{M}(\rho) = (\mathcal{E}, \lambda, \mathsf{pid}, \delta, \rightarrow, \rhd)$ is defined as follows. $\mathcal{M}(\rho)$

The set of events is $\mathcal{E} = \{1, 2, \dots, n\}$. For $1 \le x \le n$, we set $\mathsf{pid}(x) = p_x$ and $\lambda(x) = a_x$. For $1 \le x \le n$, let τ_x be the transition taken for moving from c_{x-1} to c_x. Note that τ_x is uniquely determined by 1) $\mathsf{pid}(x)$, 2) the control locations corresponding to $\mathsf{pid}(x)$ in c_{x-1} and c_x and 3) change in the data-structure contents of c_{x-1} and c_x, if any. We set $\delta(x) = d$ if $\tau_x \in \mathrm{Trans}_{p_x \rightarrow d} \cup \mathrm{Trans}_{p_x \leftarrow d}$.

The relation $\rightarrow$ is uniquely determined by the pid mapping and the total order on the sequence of configurations. For $x, y \in \mathcal{E}$, $x \rightarrow y$ if $\mathsf{pid}(x) = \mathsf{pid}(y)$, say p, and $x < y$ and for all z such that $x < z < y$, $\mathsf{pid}(z) \neq p$.

For $0 \le x \le n$, let $c_x = ((\ell_p^x)_{p \in \mathbf{Procs}}, (u_d^x)_{d \in \mathbf{DS}})$. We will recover the $\rhd$-relation for each data-structure $d \in \mathbf{DS}$.

Let $d \in \mathbf{Stacks}$ be a stack data-structure. The nesting relation $\rhd^d$ is given by $\{(x, y) \mid u_d^{x-1} = u, u_d^x = u\ell, u_d^{y-1} = u\ell$ and $u_d^y = u$ and for all z such that $x < z < y$, $u_d^z \neq u\}$. Note that u_d^x denotes the contents of the stack d at configuration x.

Similarly for a queue data-structure $d \in \mathbf{Queues}$, the message relation $\rhd^d$ is given by $\{(x, y) \mid u_d^{x-1} = u$, $u_d^x = u\ell$, $u_d^{y-1} = \ell u'$ and $u_d^y = u'$ and the number of reads on queue d between c_x and c_y is $|u|\}$ where u_d^x denotes the contents of the queue d at configuration x.

By virtue of the stack and queue access policies, the induced nesting relations follow LIFO and the message relations follow FIFO. Thus the resulting structure is an MSCN.

Now, we define the 'sequential' language of a CPDS $\mathcal{S}$ as the set of MSCNs associated to accepting sequential runs of the CPDS. That is, $\mathscr{L}_{\mathrm{seq}}(\mathcal{S}) = \{\mathcal{M}(\rho) \mid \rho$ is an accepting sequential run of $\mathcal{S}\}$. $\mathscr{L}_{\mathrm{seq}}(\mathcal{S})$

Theorem 2.9. $\mathscr{L}_{\mathrm{po}}(\mathcal{S}) = \mathscr{L}_{\mathrm{seq}}(\mathcal{S})$.

$\mathscr{L}(\mathcal{S})$ Henceforth, it is denoted $\mathscr{L}(\mathcal{S})$.

Proof. ($\supseteq$) Let $\mathcal{M}(\rho) \in \mathscr{L}_{\mathrm{seq}}(\mathcal{S})$. An accepting po-run (c-loc, d-loc) is uniquely determined by ρ. Recall that τ_x is the transition taken for moving from c_{x-1} to c_x, and $\mathsf{pid}(x) = p_x$. Then c-loc(x) is indeed the target control location of τ_x. That is, c-loc$(x) = \ell_{p_x}^x$. Also

$$\text{d-loc}(x) = \begin{cases} \ell & \text{if } \tau_x \text{ is of the form } (\ell_{p_x}, a, \ell'_{p_x}, \ell) \in \mathrm{Trans}_{p_x \rightarrow d} \\ \bot & \text{otherwise.} \end{cases}$$

Claim 2.10. (c-loc, d-loc) *is an accepting* po-run *of* $\mathcal{S}$ *on* $\mathcal{M}(\rho)$.

($\subseteq$) Let $\mathcal{S}$ be a CPDS and $\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \delta, \rightarrow, \rhd)$ be an MSCN. Further, let (c-loc, d-loc) be a po-run of $\mathcal{S}$ on $\mathcal{M}$, and let $<$ be a linearisation[1] of the events of $\mathcal{M}$.

[1]A linearisation of a partial order is a total order extension of it. In the case of MSCNs, a linearisation is a total order which respects $(\rightarrow \cup \rhd)^*$ order.

We can associate a seq-run ρ to the tuple $(\mathcal{M}, (\text{c-loc}, \text{d-loc}), <)$ as follows:

We associate a configuration to each prefix of the MSCN with respect to the linearisation $<$.

A prefix is a downward closed set $\mathcal{E}' \subseteq \mathcal{E}$. The *frontier* of a prefix is the tuple of maximal events of $\mathcal{E}'$ with respect to $\rightarrow$. That is, the frontier of $\mathcal{E}'$ is $(e_1, \dots e_{\mathtt{p}})$ if for each p, either $e_p \in \mathcal{E}'$ and $\mathsf{pid}(e_p) = p$ and there does not exist $f \in \mathcal{E}'$ such that $e \rightarrow f$, or there is no $e \in \mathcal{E}'$ such that $\mathsf{pid}(e) = p$ and $e_p = \bot_p \notin \mathcal{E}$.

The configuration associated to a prefix $\mathcal{E}'$ is denoted $c_{\mathcal{E}'}$. The set of control locations of $c_{\mathcal{E}'}$ is given by $\text{c-loc}(e_1), \dots, \text{c-loc}(e_{\mathtt{p}})$ where $(e_1, \dots, e_{\mathtt{p}})$ is the frontier of $\mathcal{E}'$. Recall that by convention $\text{c-loc}(\bot_p) = \ell_p^{\text{in}}$.

The contents of the data-structure d is $u_d = \text{d-loc}(e_1) \dots \text{d-loc}(e_x)$ if $e_1 \rightarrow^+ e_2 \rightarrow^+ \dots e_x$ is the *maximal* sequence of pending d-writes in $\mathcal{E}'$. An event e is a pending d-write in $\mathcal{E}'$ if $e \rhd f$ for some $f \notin \mathcal{E}'$ and $\delta(e) = d$.

The sequential run $\text{po-to-seq}(\mathcal{M}, (\text{c-loc}, \text{d-loc}), <)$ of $\mathcal{S}$ is defined to be

$$\rho = c_{\mathcal{E}^0} \xrightarrow{a_1}_{p_1} c_{\mathcal{E}^1} \xrightarrow{a_2}_{p_2} c_{\mathcal{E}^2} \dots \xrightarrow{a_n}_{p_n} c_{\mathcal{E}^n}$$

where

- $\emptyset = \mathcal{E}^0 \subseteq \mathcal{E}^1 \subseteq \dots \mathcal{E}^n = \mathcal{E}$ are the prefixes induced by the linearisation $<$.
- Let e_j be the jth event according to the linearization $<$. Then, $a_j = \lambda(e_j)$ and $p_j = \mathsf{pid}(e_j)$.

Claim 2.11. *ρ is an accepting* seq-run *of $\mathcal{S}$ on $\mathcal{M}$. Furthermore, $\mathcal{M}(\rho) = \mathcal{M}$.*

□

2.4.3 Closure properties

The language of a CPDS is a set of MSCNs. The class of MSCNs recognised by a CPDS enjoys several closure properties. These are closed under union, intersection, concatenation and alphabetic projection (non-erasing homomorphism).

Theorem 2.12. *The class of languages recognised by CPDS is closed under union, intersection, concatenation and renaming.*

Proof. We can effectively obtain the CPDS for each of these operations using standard constructions. We detail the construction for union, and briefly discuss the other cases.

For closure under union, we may assume that the control locations of each CPDS are disjoint. The set of control locations for the union will be the union of the control locations of each together with a new initial control location. On the initial transition, the CPDS non-deterministically chooses to simulate one of the two CPDS. The global acceptance condition makes sure that the local non-deterministic guess made at the initial transition is consistent among all the processes. The global accepting set is the union of both.

To define the CPDS for union more formally, let $\mathcal{S}_1 = (\mathsf{Locs}^1, \text{Trans}^1, (\ell_1^1, \ldots, \ell_{\mathfrak{p}}^1), \mathsf{Locs}_{\text{fin}}^1)$ and $\mathcal{S}_2 = (\mathsf{Locs}^2, \text{Trans}^2, (\ell_1^2, \ldots, \ell_{\mathfrak{p}}^2), \mathsf{Locs}_{\text{fin}}^2)$ be two CPDS. We may assume $\mathsf{Locs}^1 \cap \mathsf{Locs}^2 = \emptyset$. The CPDS for the union $\mathcal{S} = (\mathsf{Locs}, \text{Trans}, \underbrace{(\ell_{\text{in}}, \ldots, \ell_{\text{in}})}_{\mathfrak{p}}, \mathsf{Locs}_{\text{fin}})$ where

- $\mathsf{Locs} = \mathsf{Locs}^1 \cup \mathsf{Locs}^2 \cup \{\ell_{\text{in}}\}$ where $\ell_{\text{in}} \notin \mathsf{Locs}_1 \cup \mathsf{Locs}_2$.
- The global initial state is $\underbrace{(\ell_{\text{in}}, \ldots, \ell_{\text{in}})}_{\mathfrak{p}}$.
- The set of global final states is the union of both: $\mathsf{Locs}_{\text{fin}} = \mathsf{Locs}_{\text{fin}}^1 \cup \mathsf{Locs}_{\text{fin}}^2$.
- The set of transition Trans is again a tuple $(\text{Trans}_p)_{(p \in \mathbf{Procs})}$. The local transitions on process p contains those of both: $\text{Trans}_p^1 \subseteq \text{Trans}_p$ and $\text{Trans}_p^2 \subseteq \text{Trans}_p$. In addition Trans_p also contains the following initial transitions, where $b \in \{1, 2\}$ and ℓ_p^b is the local initial state of process p in $\mathcal{S}_b$:

$$\begin{aligned}
(\ell_{\text{in}}, a, \ell) \qquad & \text{if } (\ell_p^b, a, \ell) \in \text{Trans}_p^b \\
(\ell_{\text{in}}, a, \ell, \ell') \qquad & \text{if } (\ell_p^b, a, \ell, \ell') \in \text{Trans}_p^b \\
(\ell_{\text{in}}, \ell, a, \ell') \qquad & \text{if } (\ell_p^b, \ell, a, \ell') \in \text{Trans}_p^b
\end{aligned}$$

Closure under intersection similarly follows the standard constructions. The cartesian product of the two CPDS gives the automaton for intersection. The number of control locations in the product automaton is the product of the number of control locations in each.

Concatenation of two MSCNs over the same architecture concatenates them process-wise. That is, for each process p, it adds a $\rightarrow$ edge from the maximum event on p of the first MSCN to the minimum event on p of the second MSCN. This, can be lifted to sets of MSCNs as well. CPDs are closed under concatenation. This means that, given two CPDS $\mathcal{S}_1$ and $\mathcal{S}_2$ over an architecture $\mathfrak{A}$, we can effectively construct a CPDS $\mathcal{S}$ such that $\mathscr{L}(\mathcal{S}) = \mathscr{L}(\mathcal{S}_1) \cdot \mathscr{L}(\mathcal{S}_2)$.

We first consider the case where $\mathcal{S}_1$ has only one global accepting state. In this case, the CPDS $\mathcal{S}$ is obtained by the standard constructions from $\mathcal{S}_1$ and $\mathcal{S}_2$. The initial state is inherited from $\mathcal{S}_1$ and the final states fro $\mathcal{S}_2$; the initial transitions of $\mathcal{S}_2$ are tweaked in to behave as if global accepting state of $\mathcal{S}_1$ are the global initial state of $\mathcal{S}_2$.

Now, for the general case where $\mathcal{S}_1$ has several global accepting states, it can be written as the union of several CPDS with a single global accepting state. Since CPDS are closed under union, and since $(\mathscr{L}_1 \cup \mathscr{L}_2) \cdot \mathscr{L}_3 = \mathscr{L}_1 \cdot \mathscr{L}_3 \cup \mathscr{L}_2 \cdot \mathscr{L}_3$, closure under concatenation follows.

Closure under renaming can also be shown using standard techniques. On each letter, the CPDS guesses a transition on its pre-image. No additional control locations are needed. □

From the closure under union, we get the following corollary:

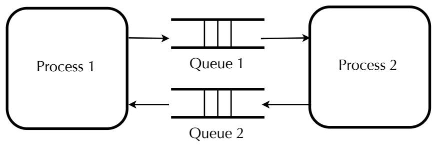
Corollary 2.13. *The model of CPDS which allows sets of global initial states is no more powerful than the one with a single global initial state.*

Remark 2.14. We choose to keep a single global initial state in the definition of CPDS as it makes some technical proofs simpler.

The class of languages of MSCNs recognisable by CPDS is not closed under complementation. This follows from a result due to Bollig and Leucker [BL06, Bol05] that simpler systems of communicating finite state machines are not closed under complementation.

Theorem 2.15 ([BL06, Bol05])**.** *The class of languages recognisable by CPDS over $\mathfrak{A}$ and Σ are not closed under complementation, as soon as $\mathfrak{A}$ embeds the following architecture.*

Procs	=	$\{1, 2\}$
Stacks	=	$\emptyset$
Queues	=	$\{q_1, q_2\}$
Writer	=	$\{q_1 \mapsto 1, q_2 \mapsto 2\}$
Reader	=	$\{q_1 \mapsto 2, q_2 \mapsto 1\}$



The class of deterministic CPDS is weaker than the class of non-deterministic CPDS. More precisely, a non-deterministic CPDS is not determinisable always. In fact, the non-determinisability is a consequence of non-complementability: If CPDS were determinisable, then they could be complemented by complementing the global accepting states.

Corollary 2.16. *The class of languages recognised by deterministic CPDS is strictly contained in that recognised by CPDS.*

2.5 Specification formalisms

We will see three different and powerful specification formalisms for MSCNs here: Monadic Second Order Logic which serves as a classical logic, Propositional Dynamic Logic which serves as a navigational logic, and Temporal Logics.

2.5.1 MSO over MSCNs

Monadic Second Order Logic is a very expressive classical logic. We assume an infinite supply of first-order variables $x, y, \ldots$ and second-order variables $X, Y, \ldots$. First order variables vary over events of an MSCN while second order variables vary over subsets of events. The vocabulary of MSO is fixed by the architecture $\mathfrak{A}$ and the set of actions Σ. The syntax of $\mathsf{MSO}(\mathfrak{A}, \Sigma)$ is as follows: $\mathsf{MSO}(\mathfrak{A}, \Sigma)$

$$\varphi ::= a(x) \mid p(x) \mid d(x) \mid x \to y \quad \mid x = y \mid x \in X \mid \varphi \vee \varphi \mid \neg\varphi \mid \exists x\, \varphi \mid \exists X\, \varphi$$

where $p \in \mathbf{Procs}$, $a \in \Sigma$, and $d \in \mathbf{DS}$.

An $\mathsf{MSO}(\mathfrak{A}, \Sigma)$ formula is evaluated over an MSCN $\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \delta, \to, \rhd) \in \mathbb{MSCN}(\mathfrak{A}, \Sigma)$. The semantics assigns truth values for a formula under a valuation of free variables. A valuation of free variables assigns events to the first order variables, and sets of events to the second order variables. We denote a valuation by ν. $\nu[x \mapsto e]$ represents a valuation which agrees with ν on all variables except x, which is mapped to e.

$\mathcal{M}, \nu \models a(x)$	if $\lambda(\nu(x)) = a \in \Sigma$
$\mathcal{M}, \nu \models p(x)$	if $\mathsf{pid}(\nu(x)) = p \in \mathbf{Procs}$
$\mathcal{M}, \nu \models d(x)$	if $\delta(\nu(x)) = d \in \mathbf{DS}$
$\mathcal{M}, \nu \models x \to y$	if $(\nu(x), \nu(y)) \in \to$
$\mathcal{M}, \nu \models x \rhd y$	if $(\nu(x), \nu(y)) \in \rhd$
$\mathcal{M}, \nu \models x = y$	if $\nu(x) = \nu(y)$
$\mathcal{M}, \nu \models x \in X$	if $\nu(x) \in \nu(X)$
$\mathcal{M}, \nu \models \varphi \vee \psi$	if $\mathcal{M}, \nu \models \varphi$ or $\mathcal{M}, \nu \models \psi$
$\mathcal{M}, \nu \models \neg\varphi$	if it is not the case that $\mathcal{M}, \nu \models \varphi$
$\mathcal{M}, \nu \models \exists x\, \varphi$	if there exists $e \in \mathcal{E}$ such that $\mathcal{M}, \nu[x \mapsto e] \models \varphi$
$\mathcal{M}, \nu \models \exists X\, \varphi$	if there exists $\mathcal{E}' \subseteq \mathcal{E}$ such that $\mathcal{M}, \nu[X \mapsto \mathcal{E}'] \models \varphi$

Macros We may also use macros: $\varphi_1 \wedge \varphi_2$ for $\neg(\neg\varphi_1 \vee \neg\varphi_2)$; $\forall x \varphi$ for $\neg\exists x \neg\varphi$; $\forall X \varphi$ for $\neg\exists X \neg\varphi$; $\varphi_1 \implies \varphi_2$ for $\varphi_2 \vee \neg\varphi_1$; $x \rhd^d y$ for $d(x) \wedge x \rhd y$.

The partial order on MSCNs is expressible in MSO. We denote the partial order by $\leq$ and use it as a macro: $x \leq y$ stands for $\forall X\, ((\forall z_1\, \forall z_2\, (z_1 \in X \wedge (z_1 \to z_2 \vee z_1 \rhd z_2)) \implies z_2 \in X)) \implies (x \in X \implies y \in X)$.

The linear order on the processes (or $(\to)^*$) is denoted $\leq_{\text{proc}}$. $x \leq_{\text{proc}} y$ stands for $\forall X\, ((\forall z_1\, \forall z_2\, (z_1 \in X \wedge z_1 \to z_2) \implies z_2 \in X)) \implies (x \in X \implies y \in X)$.

Now that we have the partial order $\leq$, we can also have a macro for the "concurrent" relation: $\mathsf{concurrent}(x, y)$ stands for $\neg x \leq y \wedge \neg y \leq x$

Example 2.17. Let us state the property: every a labelled event has at least another a labelled event which is concurrent to it.

$$\forall x\, a(x) \implies \exists y\, \mathsf{concurrent}(x, y) \wedge a(y)$$

Example 2.18. Consider the following safety property:

An event labelled b does not occur in the scope of a function a.

This means that, there is an event labelled b during which the program stack should have some contents pushed on a call to function a (abstracted by the action label a). This can be expressed in MSO as:

$$\neg\exists x \exists y \exists z\, (b(x) \wedge (y \rhd z \wedge y \leq_{\text{proc}} x \wedge x \leq_{\text{proc}} z \wedge a(x) \wedge \bigvee_{d \in \mathrm{Stack}} d(x))$$

We give a couple of scenarios where this property is relevant:

- The function a is secure: During a secure function no communication to neighbouring processes is allowed. For this, we may replace 'label b' with 'an access to a queue data-structure' (that is, $b(x)$ is replaced with $\bigvee_{d \in \mathbf{Queues}} d(x)$).
- Recall that each process of a CPDS may be internally multi-threaded (models interleaving behaviour of several recursive programs by several stacks). Suppose a is an important function that needs to be executed interrupt free. Thus during the scope of a, the processor must execute b — here b stands for a big disjunction over the accesses to other stacks and local actions by other threads.
- In [LMP08a], a 'well-queuing' assumption was required to obtain the decidability for reachability of communicating recursive programs. The well-queuing property states that local stacks must be empty when a process reads from a queue. This is same as stating that all functions are secure.

Existential Monadic Second-Order Logic ($\mathsf{EMSO}(\mathfrak{A}, \Sigma)$) This is a fragment of $\mathsf{MSO}(\mathfrak{A}, \Sigma)$ which employs only existentially quantified second-order variables, which appears as a prefix of the formula. $\mathsf{EMSO}(\mathfrak{A}, \Sigma)$

Thus a formula $\varphi \in \mathsf{MSO}(\mathfrak{A}, \Sigma)$ is in $\mathsf{EMSO}(\mathfrak{A}, \Sigma)$ if φ is of the form $\exists X_1 \exists X_2 \ldots \exists X_n \psi$ where ψ does not use a second-order quantification.

First-Order Logic ($\mathsf{FO}(\mathfrak{A}, \Sigma)$) This is a weaker fragment of $\mathsf{MSO}(\mathfrak{A}, \Sigma)$ which does not use any second-order quantification. $\mathsf{FO}(\mathfrak{A}, \Sigma)$

Thus a formula $\varphi \in \mathsf{MSO}(\mathfrak{A}, \Sigma)$ is in $\mathsf{FO}(\mathfrak{A}, \Sigma)$ if φ does not use any second-order quantification.

Remark 2.19. The language of a CPDS $\mathcal{S}$ over $\mathfrak{A}$ and Σ as a set of MSCNs can be described in $\mathsf{EMSO}(\mathfrak{A}, \Sigma)$. The $\mathsf{EMSO}(\mathfrak{A}, \Sigma)$ formula will employ second-order variables to guess the (c-loc, d-loc) labelling of an event. Thus there are twice as many second-order variables as the number of control locations: $\{X_\ell\}_{\ell \in \mathsf{Locs}}$ used to identify the c-loc mapping and $\{Y_\ell\}_{\ell \in \mathsf{Locs}}$ used to identify the d-loc mapping. These are existentially quantified.

A satisfying valuation function corresponding to an accepting run will assign to the second-order variable X_ℓ all the events which are assigned the control location ℓ by c-loc mapping. Similarly, the variable Y_ℓ will be assigned all the events which are mapped to the control location ℓ by d-loc mapping. The formula will then verify that for each event, the control state assignment of its local neighbourhood (assumed by the existential second order variables) respects the transition relations. The labelling of the minimal events must conform to the initial state, and those of the maximal events must conform to one of the global final states.

Let $\mathsf{EMSO}(\mathfrak{A}, \Sigma)$ formula for $\mathcal{S}$ be $\psi_{\mathcal{S}}$. One can prove that for all MSCNs $\mathcal{M} \in \mathbb{MSCN}(\mathfrak{A}, \Sigma)$, $\mathcal{M} \in \mathscr{L}(\mathcal{S})$ if and only if there exists a valuation ν such that $\mathcal{M}, \nu \models \psi_{\mathcal{S}}$. Since $\psi_{\mathcal{S}}$ does not have any free variables, ν can be any valuation, and in particular the empty one.

2.5.2 Propositional Dynamic Logic (PDL)

Propositional Dynamic Logic is a navigational logic. It allows to walk along an MSCN traversing the edges (path formulas) and verifying properties (state formulas) along the path.

A state formula in PDL is a boolean combination of atomic propositions and existence of paths. A path formula follows a path as dictated by a regular expression over edges while checking state formulas at the nodes if needed. We extend the PDL introduced in [BKM10] for message passing systems to our needs. The syntax of state formulas (σ) and path formulas (π) of $\mathsf{PDL}(\mathfrak{A}, \Sigma)$ is given by

$$\begin{aligned}
\sigma &::= \top \mid p \mid a \mid d \mid \sigma_1 \vee \sigma_2 \mid \neg\sigma \mid \langle\pi\rangle\sigma \\
\pi &::= ¿\sigma? \mid \to \mid \rhd \mid (\to)^{-1} \mid \rhd^{-1} \mid \pi_1 + \pi_2 \mid \pi_1 \cdot \pi_2 \mid \pi^*
\end{aligned}$$

where $p \in \mathbf{Procs}$, $a \in \Sigma$, and $d \in \mathbf{DS}$.

A $\mathsf{PDL}(\mathfrak{A}, \Sigma)$ formula is evaluated on an MSCN $\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \delta, \to, \rhd) \in \mathbb{MSCN}(\mathfrak{A}, \Sigma)$. The semantics of a state formula is the set of events which satisfy the formula. The semantics of a path formula is the set of pairs of events between which there is a satisfying path. The semantics is denoted by $[\![-]\!]_{\mathcal{M}}$. We may simply write $[\![-]\!]$ instead of $[\![-]\!]_{\mathcal{M}}$ id $\mathcal{M}$ is understood from the context.

$$\begin{aligned}
[\![\top]\!] &= \mathcal{E} \\
[\![p]\!] &= \{e \in \mathcal{E} \mid \mathsf{pid}(e) = p\} \\
[\![a]\!] &= \{e \in \mathcal{E} \mid \lambda(e) = a\} \\
[\![d]\!] &= \{e \in \mathcal{E} \mid \delta(e) = d\} \\
[\![\sigma_1 \vee \sigma_2]\!] &= [\![\sigma_1]\!] \cup [\![\sigma_2]\!] \\
[\![\neg\sigma]\!] &= \mathcal{E} \setminus [\![\sigma]\!] \\
[\![\langle\pi\rangle\sigma]\!] &= \{e \in \mathcal{E} \mid \exists f \in [\![\sigma]\!] \text{ such that } (e, f) \in [\![\pi]\!]\} \\
[\![¿\sigma?]\!] &= \{(e, e) \mid e \in [\![\sigma]\!]\} \\
[\![\rightarrow]\!] &= \rightarrow \\
[\![\rhd]\!] &= \rhd \\
[\![(\rightarrow)^{-1}]\!] &= [\![\rightarrow]\!]^{-1} \\
[\![(\rhd)^{-1}]\!] &= [\![(\rhd)]\!]^{-1} \\
[\![\pi_1 + \pi_2]\!] &= [\![\pi_1]\!] \cup [\![\pi_2]\!] \\
[\![\pi_1 \cdot \pi_2]\!] &= \{(e, f) \in \mathcal{E}^2 \mid \exists e_1 \in \mathcal{E} \text{ such that } (e, e_1) \in [\![\pi_1]\!] \text{ and } (e_1, f) \in [\![\pi_2]\!]\} \\
[\![\pi^*]\!] &= \{(e, f) \in \mathcal{E}^2 \mid e = f \text{ or } \exists e_1, \ldots e_n \in \mathcal{E} \text{ such that for all } 0 \le i \le n \\
&\qquad (e_i, e_{i+1}) \in [\![\pi]\!] \text{ where } e_0 = e \text{ and } e_{n+1} = f\}
\end{aligned}$$

Macros We may use macros in PDL also: $\varphi_1 \wedge \varphi_2$ for $\neg(\neg\varphi_1 \vee \neg\varphi_2)$; $\varphi_1 \implies \varphi_2$ for $\varphi_2 \vee \neg\varphi_1$; $\rhd^d$ for $¿d? \cdot \rhd \cdot ¿d?$ etc.

The partial order on MSCNs is easily expressible in PDL. We denote the partial order by $\le$ and use it as a macro: $\le$ stands for $(\rhd + \rightarrow)^*$.

PDL with intersection We can also include intersection in the path formulas of the PDL. PDL with intersection over an architecture $\mathfrak{A}$ and set of actions Σ is denoted $\mathsf{PDL}^{\cap}(\mathfrak{A}, \Sigma)$. It is given by the following syntax:

PDL with intersection

$$\begin{aligned}
\sigma &::= \top \mid p \mid a \mid d \mid \sigma_1 \vee \sigma_2 \mid \neg\sigma \mid \langle\pi\rangle\sigma \\
\pi &::= ¿\sigma? \mid \rightarrow \mid \rhd \mid (\rightarrow)^{-1} \mid \rhd^{-1} \mid \pi_1 + \pi_2 \mid \pi_1 \cdot \pi_2 \mid \pi_1 \cap \pi_2 \mid \pi^*
\end{aligned}$$

where $p \in \mathbf{Procs}$, $a \in \Sigma$, and $d \in \mathbf{DS}$.

The semantics of intersection is as expected:

$$[\![\pi_1 \cap \pi_2]\!] = [\![\pi_1]\!] \cap [\![\pi_2]\!]$$

This allows to express more properties of the system. However, as we will see later, the complexity of the decision procedures will be one exponential higher.

Example 2.20. Consider Example 2.18. The property can be described in $\mathsf{PDL}^{\cap}$ as follows:

$$a \wedge \langle \rhd \cap \rightarrow^* \cdot ¿b? \cdot \rightarrow^* \rangle \top$$

`sentence` We introduce *sentences* as there are no natural starting points for MSCNs [DG06]. A $\mathsf{PDL}(\mathfrak{A}, \Sigma)$ (resp. $\mathsf{PDL}^{\cap}(\mathfrak{A}, \Sigma)$) sentence ϕ is given by:

$$\phi = \top \mid \mathsf{E}\,\sigma \mid \phi \vee \phi \mid \neg\phi$$

where σ is a $\mathsf{PDL}(\mathfrak{A}, \Sigma)$ (resp. $\mathsf{PDL}^{\cap}(\mathfrak{A}, \Sigma)$) state formula. The sentence $\mathsf{E}\,\sigma$ holds true in an MSCN $\mathcal{M}$ if the semantics of σ in $\mathcal{M}$ is non-empty. That is,

$$\mathcal{M} \models \mathsf{E}\,\sigma \text{ if } [\![\sigma]\!]_{\mathcal{M}} \neq \emptyset$$

The semantics of the boolean connectives are as expected.

2.5.3 Temporal Logics

We propose a temporal logic over MSCNs in the spirit of local temporal logics over partial orders [APP95, DG06, GK03, GK10, Thi94].

The temporal logic we propose has several ‘next’ and ‘until’ modalities. The ‘next modality intuitively allows us to move one ‘step’, and the ‘until’ modality allows us to follow a sequence of (guarded) ‘steps’ in order to meet a requirement. We allow several ‘steps’ in our syntax, which are essentially combinations of basic edge relations of the graph, and their converses.

The syntax of $\mathsf{TL}(\mathfrak{A}, \Sigma)$ of local temporal logics is as follows:

$$\varphi = a \mid p \mid d \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathsf{X}_{\pi}\varphi \mid \varphi\, \mathsf{U}_{\pi}\, \varphi$$

where $a \in \Sigma$, $p \in \mathbf{Procs}$, $d \in \mathbf{DS}$ and π could be one among $\{\rightarrow, \rhd, \rightarrow^{-1}, \rhd^{-1}, \rightarrow + \rhd, \rightarrow^{-1} + \rhd^{-1}, \rightarrow + \rhd + \rightarrow^{-1} + \rhd^{-1}\}$.

Thus $\mathsf{TL}(\mathfrak{A})$ has both future and past modalities. The modalities $\mathsf{X}_{\rightarrow}$ and $\mathsf{X}_{\rhd}$ allow to reason about the successor events in the graph. The past counterparts $\mathsf{X}_{\rightarrow^{-1}}$ and $\mathsf{X}_{\rhd^{-1}}$ allow to reason about the predecessor events. The until modality U_{π} asks for the existence of a π-path until an event which satisfies a requirement, and in addition makes sure that the events along this path satisfy another criterion.

The temporal logic formulas are evaluated on the events of an MSCN $\mathcal{M} =$

$(\mathcal{E}, \lambda, \mathsf{pid}, \delta, \rightarrow, \rhd)$.

$$\begin{array}{ll}
\mathcal{M}, e \models a & \text{if } \lambda(e) = a \in \Sigma \\
\mathcal{M}, e \models p & \text{if } \mathsf{pid}(e) = p \in \mathbf{Procs} \\
\mathcal{M}, e \models d & \text{if } \delta(e) = d \in \mathbf{DS} \\
\mathcal{M}, e \models \neg\varphi & \text{if it is not the case that } \mathcal{M}, e \models \varphi \\
\mathcal{M}, e \models \varphi_1 \vee \varphi_2 & \text{if } \mathcal{M}, e \models \varphi_1 \text{ or } \mathcal{M}, e \models \varphi_2 \\
\mathcal{M}, e \models \mathsf{X}_\pi \varphi & \text{if there exists } f \in \mathcal{E} \text{ such that } (e, f) \in [\![\pi]\!] \text{ and } \mathcal{M}, f \models \varphi \\
\mathcal{M}, e \models \varphi_1 \,\mathsf{U}_\pi\, \varphi_2 & \text{if there is a sequence } e = e_0, e_1, e_2 \ldots e_{n-1}, e_n \\
 & \text{such that } (e_i, e_{i+1}) \in [\![\pi]\!] \text{ for all } 0 \le i < n \text{ and} \\
 & \mathcal{M}, e_i \models \varphi_1 \text{ for each } 0 \le i < n \text{ and } \mathcal{M}, e_n \models \varphi_2
\end{array}$$

Example 2.21. Consider the property: If the b (for 'begin') signal is activated on an access to data-structure d by process p, then the process p is dedicated to the data-structure d until an e (for 'end') signal is activated. This can be said by the following TL formula:

$$p \wedge d \wedge b \implies (\bigwedge_{d' \neq d} \neg d') \,\mathsf{U}_{\rightarrow}\, (d \wedge e)$$

All the modalities of $\mathsf{TL}(\mathfrak{A}, \Sigma)$ can be written equivalently in $\mathsf{PDL}(\mathfrak{A}, \Sigma)$ (Table 2.3). We denote this translation by $\mathsf{TL2PDL}$. Note that $\mathcal{M}, e \models \varphi$ if and only if $e \in [\![\mathsf{TL2PDL}(\varphi)]\!]$.

$$\begin{array}{rcl}
\mathsf{TL2PDL}(a) & = & a \\
\mathsf{TL2PDL}(p) & = & p \\
\mathsf{TL2PDL}(d) & = & d \\
\mathsf{TL2PDL}(\neg\varphi) & = & \neg\mathsf{TL2PDL}(\varphi) \\
\mathsf{TL2PDL}(\varphi_1 \vee \varphi_2) & = & \mathsf{TL2PDL}(\varphi_1) \vee \mathsf{TL2PDL}(\varphi_2) \\
\mathsf{TL2PDL}(\mathsf{X}_\pi \varphi) & = & \langle \pi \rangle \mathsf{TL2PDL}(\varphi) \\
\mathsf{TL2PDL}(\varphi_1 \,\mathsf{U}_\pi\, \varphi_2) & = & \langle (¿\mathsf{TL2PDL}(\varphi_1)? \cdot \pi)^* \rangle \mathsf{TL2PDL}(\varphi_2)
\end{array}$$

Table 2.3: Translation from $\mathsf{TL}(\mathfrak{A})$ to $\mathsf{PDL}(\mathfrak{A})$

For temporal logics also, we introduce sentences. A $\mathsf{TL}(\mathfrak{A}, \Sigma)$ sentence ϕ is given by:

$$\phi = \top \mid \mathsf{E}\,\varphi \mid \phi \vee \phi \mid \neg\phi$$

The sentence $\mathsf{E}\,\varphi$ holds true in an MSCN $\mathcal{M}$ if there is an event e of $\mathcal{M}$ such that $\mathcal{M}, e \models \varphi$. That is,

$$\mathcal{M} \models \mathsf{E}\,\varphi \text{ if there is an event } e \text{ of } \mathcal{M} \text{ such that} \mathcal{M}, e \models \varphi$$

The sematics of the boolean connectives are as expected.

A temporal logic sentence can be translated to a PDL sentence.

TL with arbitrary PDL steps

Remark 2.22. We can extend our temporal logic to allow unrestricted steps (π) in X_π and U_π. The results we show for $\mathsf{TL}(\mathfrak{A}, \Sigma)$ hold for this extension also, without affecting the complexity.

TL with MSO definable modalities

Remark 2.23. We can further extend our temporal logic by allowing more modalities definable in MSO. Note that all the modalities of $\mathsf{TL}(\mathfrak{A}, \Sigma)$ are expressible in $\mathsf{MSO}(\mathfrak{A}, \Sigma)$. It is desirable to have more convenient modalities. For example, we could define a 'concurrent' modality #, which allows to reason about a concurrent event. Another modality would be a 'universal' until $\varphi_1 \mathsf{U}^\forall \varphi_2$ which requires that a later event satisfies φ_2 and *all* the events in between satisfy φ_1. Note that these two modalities are not expressible in PDL.

The techniques and results presented in this thesis can be extended to capture any temporal logic definable in this generic framework of 'temporal logics with MSO definable modalities' (cf. [BCGZ11, GK10]). However this will alter the complexity.

Macros We keep the usual macros for $\mathsf{TL}(\mathfrak{A})$ as well: $\varphi_1 \wedge \varphi_2$ for $\neg(\neg\varphi_1 \vee \neg\varphi_2)$; $\varphi_1 \implies \varphi_2$ for $\varphi_2 \vee \neg\varphi_1$

Eventually (F_π) and *Always* (G_π) are defined as follows: $\mathsf{F}_\pi\varphi = \top \mathsf{U}_\pi \varphi$ and $\mathsf{G}_\pi\varphi = \neg\mathsf{F}_\pi(\neg\varphi)$.

2.5.4 Discussions

Data-Structure alphabet

The data-structure language: It is often desirable to reason about the stack-languages (or queue-languages). The stack language is the set of possible stack configurations. Similarly the queue language is the set of possible stack contents. In order to reason about them independently of the states of the system, it is desirable to have an independent stack-alphabet and queue alphabet. That is essentially to say that the stack and queue alphabets are part of the architecture, and are fixed.

We can reduce the case to our setting by enriching the action labels to include the information about the stack symbols (resp. queue symbols) as well. Thus new action labels are pairs, one field keeps track of the 'real' action label and the other stores the data-structure alphabet. The events accessing a data-structure must be labelled by such pairs. Of course, the 'data-structure field' of the source and target events of a $\rhd$-edge must agree.

Let $\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \delta, \rightarrow, \rhd)$ be an MSCN. Consider any prefix $\mathcal{E}' \subseteq \mathcal{E}$. $\mathcal{E}'$ is a downward-closed set, as it is a prefix. The contents of data-structure d at prefix $\mathcal{E}$ is the sequence $c(e_1)c(e_2)\ldots c(e_n)$ where c_e corresponds to the contents of the 'data-structure field' of event e, and $e_1, \ldots, e_n$ is a maximal (ordered) subsequence of open writes to d with respect to $\mathcal{E}'$. That is, for all $j \in \{1, \ldots, n\}$, $\delta(e_j) = d$ and there exists $e' \in \mathcal{E} \setminus \mathcal{E}'$ such that $e_j \rhd e'$. Of course, $\mathsf{pid}(e_j) = \mathsf{Writer}(d)$ and $e_1, \ldots e_n$ are ordered by the $\rightarrow$ ordering.

Equipped like this, our specification formalisms permit us to reason about properties of data-structure languages. Any regular property of data-structure language can be expressed in MSO. It suffices to consider all prefixes of the MSCN and then to relativize the regular property to the "open" $\rhd$-positions.

With PDL, however, identifying the "open" $\rhd$-positions is challenging. For a stack data-structure d it can be achieved by allowing to skip over some $\rhd$-edges. For queue-data structure the "open" $\rhd$-positions cannot be identified using the intersection-free fragment. Once the "open" $\rhd$-positions are identified, PDL can also reason about regular properties of these languages, as the path expressions can verify the regular expressions along the path.

2.6 Specific Architectures

We have seen a generic framework for modelling the system with several data-structures. We also have formalism for modelling their behaviours, and specification languages to reason about them. Our generic framework captures several interesting classes of systems as special cases. Now we will have a glimpse at some well-studied specific architectures of independent interest.

Finite state machines An architecture with only one process and no data-structures is nothing but a framework for finite state machines. The MSCNs in this case are simply words.

Communicating finite state machines If the architecture has no stacks and there is at most one queue (FIFO communication channel) between every pair of processes in each direction, we have the well-studied class of communicating finite state machines [BZ83]. The MSCNs in this case are message sequence charts (MSCs). The logics we defined in Section 2.5 correspond to PDL over MSCs [BKM10] and a variant of local trace temporal logics over MSCs (TrLTL) [GK10, Thi94].

Pushdown systems If we have only one process and one stack, we are essentially having a pushdown system. Put in our framework, the CPDS over this specific architecture corresponds to nested word automata [AM09]. The MSCNs over this architecture are words with a (single) nesting relation. These are called nested words. Temporal and navigational logics over nested words have been studied in the literature [AAB$^+$08, AEM04, BCGZ11]. Our temporal logic and PDL when restricted to nested words subsume these.

Multi-pushdown systems If we have only one process and several stacks but no queues, we have a multi-pushdown system. These model the interleaving semantics of multi-threaded boolen recursive programs (without data-structures). The behaviours in this case are multiply nested words. We have introduced navigational and temporal logics for these behaviours in [BCGZ11]. Another temporal logic for these objects is studied in [LN12]. Our generic temporal logic and PDL subsume these too.

Communicating pushdown systems If we have at most one stack per process, and at most one queue between each pair of processes in each direction, we have a communicating pushdown system. This generalises pushdown system and communicating finite state machines. These systems are studied in [LMP08a, HLMS10, MP11] etc. The MSCNs in this case are called stack-queue graphs in [MP11].

2.7 Decision Problems

We have found suitable frameworks to model the system and their behaviours. We also saw formalisms to specify properties/requirements of these systems. Also we saw that this generic framework captures several interesting classes of systems. Now we will address our main goal — which is to formally verify these complex systems.

The verification method we adopt in this thesis is that of model-checking. We do model-checking on the design of our complex system, before really implementing it. After all finding a bug at the design phase saves money, time, and lives (these are safety-critical systems). Consider the following scenario:

A hypothetical firm is specialised in formally verified system design. The main service offered by this firm is, as one would guess, designing a system according to the specific requirements by a client. Their salient feature is that they also formally verify their design, and guarantee satisfaction to the client. We now describe a product development cycle in this firm.

A client gives the requirements for a distributed system to the firm. The requirements fix an architecture. Then, the engineers in the firm translate each of the requirements into a formal one, say MSO or PDL or TL, or it could even be another CPDS. Then the engineers design a system to meet the requirement. The design is again another CPDS.

Having obtained the proposed design as a CPDS and the specifications, the engineers want to verify that the design is correct. Since the designed system can be huge at times, and since a manual checking of the design is often prone to error, they would like to employ automated methods of the verification.

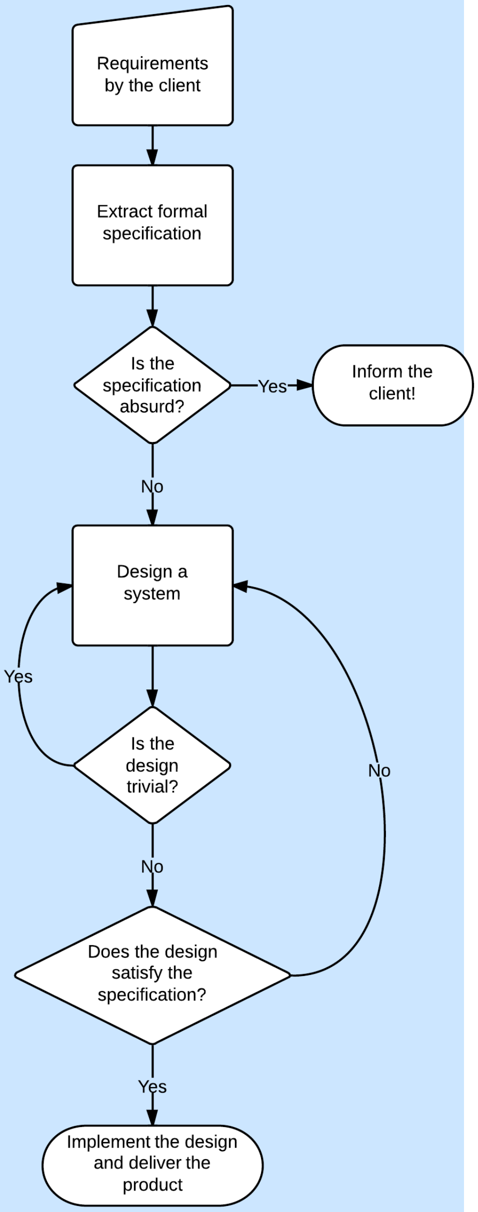
One of the very basic thing that needs to checked is whether the designed system can generate anything at all (otherwise all behaviours may vacuously satisfy the specification, but this may not please the client). This is the problem of emptiness checking. This problem can be also seen equivalently as a 'reachability question' which asks for whether a particular combination of local control locations (a combination of local control locations is a global control state) is reachable. The reachability alone can be the specification for some 'safety' criterion. One may want to verify that some 'bad/unsafe' control location is not reachable.

Another triviality check on the design is universality checking. Perhaps the design allows all possible MSCNs, and in this case, the engineers may want to reconsult their design.

Once they have verified that the design is not trivial-by-mistake, they want to verify that the formal requirements are satsified by the design. This corresponds to different problems depending on what formalism has been used to formally specify the requirement.

When the formal specification is given as another CPDS, the model-checking problem corresponds to inclusion checking. The engineers want to check whether all the behaviours of the proposed design are already contained in a 'well-behaved' system model.
They of course need to consider the model-checking problem when the specification is given in one of the logical specification formalism like MSO, PDL or TL. One important issue about the specification formalism is to be able to detect if the specification is absurd. It is very well possible that the engineers find out that their design does not meet the specification. They try to correct their design over and over, but it still does not meet the specification. It may very well be possible that this is rather due to an absurd specification which is impossible to meet. In order to rule out this undesirable situation, the engineers want to check whether the given specification is satisfiable at all. This is called the satisfiability problem.

The product-development cycle is depicted in the flowchart.


Requirements by the client
Extract formal specification
Is the specification absurd?
Yes
Inform the client!
No
Design a system
Yes
Is the design trivial?
No
No
Does the design satisfy the specification?
Yes
Implement the design and deliver the product

We will state these basic verification problems next. The architecture $\mathfrak{A}$ and the set of actions Σ are also part of the input.

Problem 1 (CPDS-Emptiness)**.**

Input $\mathcal{S}$: a CPDS over $\mathfrak{A}$ and Σ.
Question Is $\mathscr{L}(\mathcal{S}) = \emptyset$?

Problem 2 (CPDS-Inclusion)**.**
Input $\mathcal{S}_1$, $\mathcal{S}_2$: two CPDS over $\mathfrak{A}$ and Σ.
Question Is $\mathscr{L}(\mathcal{S}_1) \subseteq \mathscr{L}(\mathcal{S}_2)$?

Problem 3 (CPDS-Universality)**.**
Input $\mathcal{S}$: a CPDS over $\mathfrak{A}$ and Σ.
Question Is $\mathscr{L}(\mathcal{S}) = \mathbb{MSCN}(\mathfrak{A}, \Sigma)$?

Problem 4 (MSO-SAT)**.**
Input φ: an $\mathsf{MSO}(\mathfrak{A}, \Sigma)$ formula.
Question Does there exist $\mathcal{M} \in \mathbb{MSCN}(\mathfrak{A}, \Sigma)$ such that $\mathcal{M} \models \varphi$?

Problem 5 (PDL-SAT)**.**
Input ϕ: a $\mathsf{PDL}(\mathfrak{A}, \Sigma)$ sentence.
Question Does there exist $\mathcal{M} \in \mathbb{MSCN}(\mathfrak{A}, \Sigma)$ such that $\mathcal{M} \models \phi$?

Problem 6 (TL-SAT)**.**
Input ϕ: a $\mathsf{TL}(\mathfrak{A}, \Sigma)$ sentence.
Question Do there exist an $\mathcal{M} \in \mathbb{MSCN}(\mathfrak{A}, \Sigma)$ such that $\mathcal{M} \models \phi$?

Problem 7 (MSO-MC)**.**
Input φ: an $\mathsf{MSO}(\mathfrak{A}, \Sigma)$ formula.
$\mathcal{S}$: a CPDS over $\mathfrak{A}$ and Σ.
Question Do all $\mathcal{M} \in \mathscr{L}(\mathcal{S})$ satisfy $\mathcal{M} \models \varphi$?

Problem 8 (PDL-MC)**.**
Input ϕ: a $\mathsf{PDL}(\mathfrak{A}, \Sigma)$ sentence.
$\mathcal{S}$: a CPDS over $\mathfrak{A}$ and Σ.
Question For all $\mathcal{M} \in \mathscr{L}(\mathcal{S})$, does it hold that $\mathcal{M} \models \phi$?

Problem 9 (TL-MC)**.**
Input ϕ: a $\mathsf{TL}(\mathfrak{A}, \Sigma)$ sentence.
$\mathcal{S}$: a CPDS over $\mathfrak{A}$ and Σ.
Question For all $\mathcal{M} \in \mathscr{L}(\mathcal{S})$, does it hold that $\mathcal{M} \models \phi$?

2.7.1 Undecidability casting shadow

Our very powerful model comes with the price of undecidability. All the above verification problems are undecidable except for some specific architectures. In fact, it is so even for the special cases discussed in the previous section except finite state machines and pushdown systems.

Theorem 2.24. *The Problems 1-9 are undecidable.*

Note that the problems take the architecture $\mathfrak{A}$ and the set of actions Σ as part of the input.

In fact the undecidability holds even for fixed architecture and set of actions as soon as $|\Sigma| \geq 2$ and the architecture embeds one of the following:

$\mathfrak{A}_1$ A process with two local stacks.

$\mathfrak{A}_2$ A process with a local queue.

$\mathfrak{A}_3$ Two processes with two queues between them (the direction of the queues does not matter).

$\mathfrak{A}_4$ Two processes with two stacks and a queue between them. In this case, however, the undecidability of universality checking is not known.

The above theorem holds if the set of actions is at least binary. Any behaviour on bigger set of actions can be encoded as one on a binary set of actions. Thus, for the problems discussed here, if it is undecidable for a bigger set of actions, it is also undecidable for a binary set of actions.

Proof. The undecidability of two stack machines is a folklore result. Reachability of single process with a queue is undecidable [BZ83]. Two processes with queues between them in both directions can simulate a single process with a single queue: the second process just copies the contents from the incoming channel to the outgoing channel. Two processes with two queues between them in the same direction is also undecidable [Cha11]. Two processes with stacks communicating via a queue can check the intersection of context free languages. In fact, the undecidability holds even for reachability restricted to bounded runs [HLMS10].

The emptiness problem can be reduced to the inclusion checking, by keeping an 'empty' automaton for $\mathcal{S}_2$. Thus inclusion checking is also undecidable.

Next we argue that the satisfiability problem for $\mathsf{TL}(\mathfrak{A})$ is undecidable. We reduce the emptiness problem of CPDS to $\mathsf{TL}(\mathfrak{A})$ satisfiability. Given a CPDS $\mathcal{S}$ over $\mathfrak{A}$, we construct a $\mathsf{TL}(\mathfrak{A})$ formula $\varphi_{\mathcal{S}}$. The action labels used in $\varphi_{\mathcal{S}}$ are the set of (pairs of) control locations of $\mathcal{S}$. The formula essentially describes a run (labelling the events with states) of an MSCN. The formula $\varphi_{\mathcal{S}}$ states that 1) the labelling of the local neighbourhood of an event must conform to the transition relation, 2) the labelling of the minimal events must respect the global initial state and 3) the labelling of the maximal events must conform to one of the global accepting states.

This can be said in the unary fragment of temporal logic, in an almost pure future fragment (it needs only two past modalities $\mathsf{X}_{\rightarrow^{-1}}$ and $\mathsf{X}_{\rhd^{-1}}$). In fact, choosing transitions instead of states as action labels allows us to express valid runs in pure-future unary fragment of the logic. Thus satisfiability problem of temporal logics with pure-future and unary modalities is undecidable.

The $\mathsf{TL}(\mathfrak{A}, \Sigma)$ modalities can be expressed in $\mathsf{PDL}(\mathfrak{A}, \Sigma)$, thus $\mathsf{PDL}(\mathfrak{A}, \Sigma)$ satisfiability is also undecidable.

The unary $\mathsf{TL}(\mathfrak{A}, \Sigma)$ modalities can be expressed in $\mathsf{MSO}(\mathfrak{A}, \Sigma)$, and in fact in the first-order fragment. Thus satisfiability problem of $\mathsf{MSO}(\mathfrak{A}, \Sigma)$ and $\mathsf{FO}(\mathfrak{A}, \Sigma)$ are undecidable.

The emptiness can be reduced to $\mathsf{TL}(\mathfrak{A}, \Sigma)$-MC. The reachability condition is simply an 'eventuality' requirement. Thus $\mathsf{TL}(\mathfrak{A}, \Sigma)$-MC is undecidable. The undecidability of $\mathsf{PDL}(\mathfrak{A}, \Sigma)$-MC and $\mathsf{MSO}(\mathfrak{A}, \Sigma)$-MC follows.

For the architectures $\mathfrak{A}_1$-$\mathfrak{A}_3$, $\mathsf{FO}(\mathfrak{A}, \Sigma)$ can be effectively translated to CPDS [BL06, Bol08]. Since $\mathsf{FO}(\mathfrak{A}, \Sigma)$ is closed under complementation, the satisfiability problem of $\mathsf{FO}(\mathfrak{A}, \Sigma)$ can be reduced to the universality problem of CPDS. Given an $\mathsf{FO}(\mathfrak{A}, \Sigma)$ formula φ, consider the CPDS for its negation $\mathcal{S}_{\neg\varphi}$. The following are equivalent:

- φ is satisfiable
- $\neg\varphi$ is not valid
- $\mathcal{S}_{\neg\varphi}$ is not universal

Thus the universality problem of CPDS is undecidable. □

2.7.2 Lighting up candles

However, these classes of systems are so crucial that we do not want to give up. So we look for ways to get around undecidability.

One way is to restrict the architecture to decidable ones. In [HLMS10] and [LMP08a] the authors characterise the decidable architectures for bounded runs. However, we would like not to impose restrictions on the architecture as it is something on which we do not have much control on. We would like the architecture also to be part of the input of the verification problems.

Another technique is to consider only weaker specification formalisms, which disallow expressing undecidable problem instances. However, the undecidability results hold even with really weak fragments of specification logic, like temporal logics with only unary and future modalities, or the fragment of first order logic with only two variables. These weak fragments are not in general sufficient to express critical properties of the system. Hence we would like to keep a reasonably expressive specification language.

Another method is to adopt approximate verification. In the over-approximate verification more behaviours can be associated to the system by being more lenient about the semantics. An example is to drop the assumption of reliability of data-structures. We may assume that the data-structures may lose some entries (or data-items) non-deterministically, and this may help to obtain decidability. The study of lossy channel systems is in this spirit and is a major research line [AJ96, Cha11, Sch02].

In under-approximate verification, some restriction is imposed on the system, so that all the original behaviors need not be exhibited in the restricted one. This is also a way to regain decidability. A very well-studied under-approximation technique for communicating machines is to bound the channel sizes [GKM06, HMN$^+$05].

For multi-pushdown systems, another method of under-approximation is to impose restrictions on the access policies on stacks. The restriction called bounded context was proposed in [QR05]; more permissive bounded phase was studied in [LMP07]; a more general one with ordering on the stacks was studied in [ABH08] and [BCCCR96]; and an orthogonal restriction permitting infinite behaviours was studied in [LN11].

The under-approximate verification can also be seen as a way of parametrized verification. The under-approximation is often imposing a 'bound' on some parameter of the system, like the channel size, or the number of contexts. The complexities could be analysed wrt. these bounds. Another important parameter for under-approximate verification is tree-width, as it serves as a parameter for various fixed-parameter-tractable algorithms. In [MP11] the authors give the unified proof of decidability for various under approximate techniques studied separately otherwise via demonstrating a bound on their tree-width.

Our approach: We propose another parameter for under-approximate verification of these generic systems. This parameter, called split-width, yields decidability for the various verification problems. The parameter split-width is defined and studied in Part II.

We also study ways to implement a design so that a bound on split-width is guaranteed. Part III is devoted to this. Thus we can ensure that the behaviours of the implementation are formally verified.

Part II

Split-Width

Chapter 3

Split-width

Contents

In this chapter, we introduce the notion of split-width, which is central to this thesis.

3.1 Introduction

Split-width is a complexity measure based on an algebra for MSCNs. This algebra allows to decompose MSCNs into smaller independent parts, which can be reasoned independently. It then allows to compose the independent parts to form a bigger MSCN such that reasonings made about the smaller parts can be used to reason about the bigger MSCN. Thus it offers a divide-and-conquer approach to handle MSCNs.

We may observe that a divide-and-conquer way (or the composition / decomposition technique) of reasoning about behaviours is, in a sense, the basis of automata theory. Let us consider words. Decomposition corresponds to factorising a word and composition corresponds to concatenation.

Many foundational notions like monoid morphisms or Myhill-Nerode equivalence classes are based on decomposition / composition . An automaton or a monoid morphism on words can be seen also as a way for modular reasoning — divide a big object into simpler independent parts, abstract them independently, and obtain the abstraction for the bigger one from the abstractions of the smaller ones.

While researchers unanimously agree on factorisation / concatenation as the natural decomposition / composition technique for words, when it comes to more elaborate behavioural models, researchers have different stands on the right notion of composition / decomposition. For example, in the well-studied theory of nested words[1][AM09] the composition / decomposition used for defining congruences[AKMV05] is different from that used for obtaining an algebraic characterisation[Cyr10]. In fact the right notion of decomposition / composition in this case is a topic of debate. The practise is to choose the one that serves the purpose.

In this chapter, we propose a way to compose and decompose MSCNs for such compositional reasoning. We will also derive a measure (split-width) on MSCNs based on this decomposition technique, which provides decidability for the various parametrized verification problems.

3.2 Concurrent behaviour with matching

We consider a slight generalisation of message sequence charts with nestings for this section. We do not impose any LIFO or FIFO conditions on this matching relation $\rhd$. It is rather lenient. We only require that a node can be part of at most one $\rhd$ edge, and that the resulting graph is a partial order.

`CBM` **Definition 3.1** (Concurrent Behaviour with Matching (CBM))**.** Formally, a concurrent behaviour with matching (abbreviated as CBM) over an alphabet Σ and processes **Procs** is a tuple $\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \rightarrow, \rhd)$ where

- $\mathcal{E}$, λ, pid and $\rightarrow$ are as in Definition 2.5.
- $\rhd \subseteq \mathcal{E} \times \mathcal{E}$ is the irreflexive and disjoint matching relation: If $(e_1, e_2) \in \rhd$, $(e_3, e_4) \in \rhd$ and $(e_1, e_2) \neq (e_3, e_4)$, then $|\{e_1, e_2, e_3, e_4\}| = 4$. Moreover, if $(e_1, e_2) \in \rhd$, then $e_1 \neq e_2$.
- The underlying graph $(\mathcal{E}, (\rightarrow \cup \rhd)^*)$ is a partial order.

`Bag data-structure` Notice that, the definition of CBMs does not involve any data-structures. However, the $\rhd$ edges may be thought of as those arising from 'bag' data-structures between every pair of processes. It could also be seen as a global bag data-structure which permits writes and reads by every process. The only requirement for such a bag data-structure is that writes must be 'before' reads. This is reflected in the feature that the underlying graph is a partial order.

Example 3.2. An example of a CBM over $\Sigma = \{a, b\}$, and **Procs** $= \{1, 2\}$ is shown in Figure 3.1. The vertical arrows denote the $\rightarrow$ edges and the other arrows denote the $\rhd$ edges. Observe that the underlying graph has no cycles.

[1]These are MSCNs generated by pushdown systems.

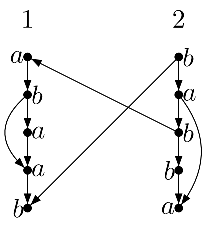


Figure 3.1: A CBM

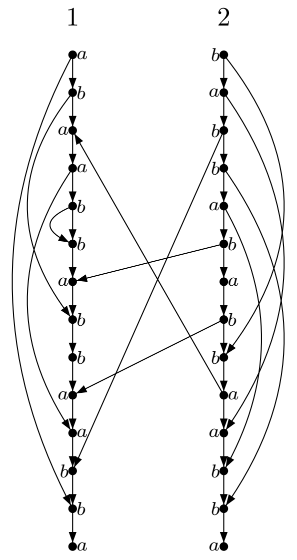


Figure 3.2: A CBM

Example 3.3. Another example of a CBM over $\Sigma = \{a, b\}$, and **Procs** $= \{1, 2\}$ is shown in Figure 3.2. Notice that the $\rhd$ edges do not follow any LIFO or FIFO policy.

Thus CBMs are a generalisation of MSCNs in the sense that the $\rhd$ relation do not have to comply to any data-structure access policies. We consider CBMs in this chapter, as our results hold for them. Indeed we obtain the results for MSCNs as a particular case of the results on CBMs.

3.3 Split concurrent behaviour with matching

In order to analyse CBMs, we consider objects called split-CBMs. As the name suggests, these are CBMs which are split: A split cuts some $\rightarrow$ edges. Thus a split-CBM is a CBM with some $\rightarrow$ edges missing. This introduces some 'holes' in the graph.

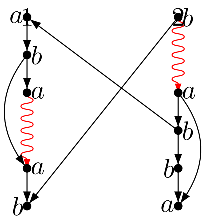


Figure 3.3: A split-CBM

We will then define an algebra on split-CBMs. This algebra allows new nodes to be inserted in the holes. However, rearranging the connected parts is disallowed. Thus, when a $\rightarrow$ edge is cut to obtain a split-CBM, it cannot be forgotten completely as the direction of this edge still needs to be conserved. This can be accomplished if the missing $\rightarrow$ edge is replaced by an 'elastic' one: more nodes could be inserted in between, but the order is maintained. Thus the non-split edges remain 'rigid' and the split-edges become 'elastic'.

Let us define split-CBMs formally:

split-CBM **Definition 3.4** (split-CBMs)**.** A *split-CBM* is a CBM together with a partitioning of its $\rightarrow$ edges into "rigid" edges (denoted $\xrightarrow{r}$) and "elastic edges" (denoted $\xrightarrow{e}$). It is a tuple $\mathcal{M\!\!\!M} = (\mathcal{E}, \lambda, \mathsf{pid}, \xrightarrow{r}, \xrightarrow{e}, \rhd)$ where $\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \rightarrow = \xrightarrow{r} \uplus \xrightarrow{e}, \rhd)$ is a CBM, which we call the *underlying CBM*.

We denote the rigid part of a split-CBM $\mathcal{M\!\!\!M}$ by $\overline{\mathcal{M}}$. That is, $\overline{\mathcal{M}} = (\mathcal{E}, \lambda, \mathsf{pid}, \xrightarrow{r}, \rhd)$. Thus a split-CBM $\mathcal{M\!\!\!M}$ can be written equivalently as $\mathcal{M\!\!\!M} = (\overline{\mathcal{M}}, \xrightarrow{e})$.

Remark 3.5. A split-CBM $\mathcal{M\!\!\!M}$ uniquely determines its rigid part $\overline{\mathcal{M}}$. However, there could be several split-CBMs yielding the same rigid part $\overline{\mathcal{M}}$.

elasticity
component
The *elasticity* of a split-CBM is the number of elastic edges it has. Thus $\mathsf{elasticity}(\mathcal{M\!\!\!M}) = |\xrightarrow{e}|$. A *component* of a split-CBM $\mathcal{M\!\!\!M}$ is a maximal connected component of $\overline{\mathcal{M}}$ wrt. $\xrightarrow{r}$ edges. In particular $\rhd$ edges and $\xrightarrow{e}$ edges are discarded when considering a maximal connected component for a component.

Example 3.6. A split-CBM is depicted in Figure 3.3, whose underlying CBM is given in Figure 3.1. The elastic edges are depicted by wavy edges. It has two elastic edges and hence it is of elasticity two. It has four components.

Example 3.7. A split-CBM over three processes is shown in Figure 3.4. It has three elastic edges, and hence elasticity three. It has six components.

Example 3.8. Another split-CBM is shown in Figure 3.5. The underlying CBM of this split-CBM is given in Figure 3.2. It has five elastic edges, and hence elasticity five. It has seven components. We call this split-CBM $\mathcal{M\!\!\!M}_{12}$ for later reference.

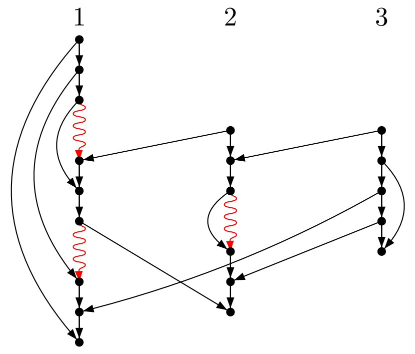


Figure 3.4: A split-CBM

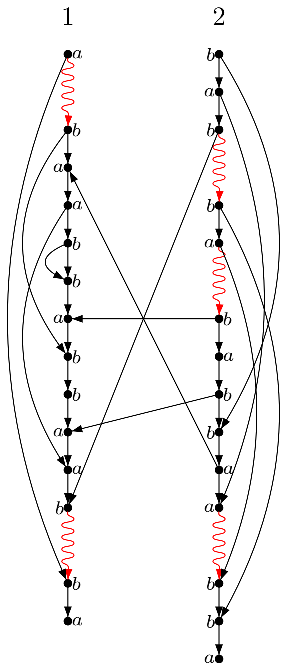


Figure 3.5: The split-CBM $\mathcal{M}_{12}$

Remark 3.9. The number of elastic edges $\xrightarrow{e}$ on a process p of a split-CBM $\mathcal{M}$ is one less than the number of components on p. Thus the number of components of a split-CBM $\mathcal{M}$ is at most $\mathsf{elasticity}(\mathcal{M}) + \mathfrak{p}$.

3.4 Merge and Shuffle

We will now define two operations on split-CBMs. 1) A unary *merge* operation which transforms an elastic edge to a rigid edge. 2) A binary *shuffle* operation that shuffles the components of two split-CBMs.

The merge operation may transform *any* of the elastic edges into a rigid one. `merge` Hence $\mathsf{merge}(\mathcal{M})$ is a set of split-CBMs. If $\mathcal{M}' \in \mathsf{merge}(\mathcal{M})$ then

- Both $\mathcal{M}$ and $\mathcal{M}'$ have the same underlying CBM.
- $\mathcal{M}'$ and $\mathcal{M}$ are the same except that, a pair of events linked by an elastic edge in $\mathcal{M}$ are linked by a rigid edge in $\mathcal{M}'$.

In other words, letting $\mathcal{M}_i = (\mathcal{E}_i, \lambda_i, \mathsf{pid}_i, \xrightarrow{r}_i, \xrightarrow{e}_i, \rhd_i)$ for $i \in \{1,2\}$, we have $\mathcal{M}_2 \in \mathsf{merge}(\mathcal{M}_1)$ if and only if:

- Both $\mathcal{M}_1$ and $\mathcal{M}_2$ have the same underlying CBM: $\mathcal{E}_1 = \mathcal{E}_2$, $\lambda_1 = \lambda_2$, $\mathsf{pid}_1 = \mathsf{pid}_2$, and $\rhd_1 = \rhd_2$. Thus, $\xrightarrow{r}_1 \uplus \xrightarrow{e}_1 = \xrightarrow{r}_2 \uplus \xrightarrow{e}_2$.
- $\xrightarrow{r}_1 \subseteq \xrightarrow{r}_2$ and $\xrightarrow{e}_2 \subseteq \xrightarrow{e}_1$.
- $|\xrightarrow{e}_1| - |\xrightarrow{e}_2| = 1 = |\xrightarrow{r}_2| - |\xrightarrow{r}_1|$.

Example 3.10. Consider the split-CBM $\mathcal{M}_{12}$ given in Example 3.8. The set of split-CBMs in $\mathsf{merge}(\mathcal{M}_{12})$ is given in Figure 3.6.

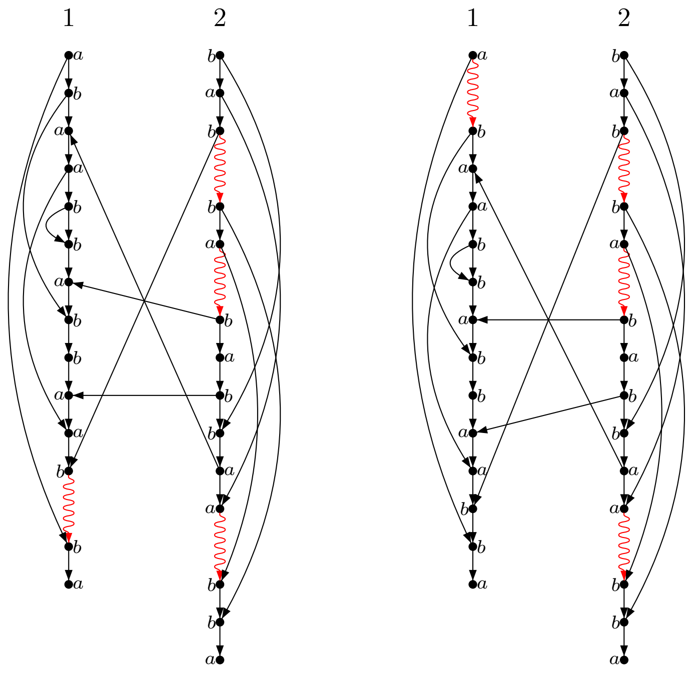
Remark 3.11. Notice that the number of components and the elasticity decrease by 1 as the result of a merge. That is, if $\mathcal{M}_2 \in \mathsf{merge}(\mathcal{M}_1)$ then $\mathsf{elasticity}(\mathcal{M}_2) = \mathsf{elasticity}(\mathcal{M}_1) - 1$. Moreover, if $\mathcal{M}_2 \in \mathsf{merge}(\mathcal{M}_1)$ and $\mathcal{M}_3 \in \mathsf{merge}(\mathcal{M}_1)$, then $\mathsf{elasticity}(\mathcal{M}_2) = \mathsf{elasticity}(\mathcal{M}_3)$. That is, the elasticity as well as the number of components of *all* split-CBMs belonging to the set $\mathsf{merge}(\mathcal{M})$ agree.

Remark 3.12. $|\mathsf{merge}(\mathcal{M})| = \mathsf{elasticity}(\mathcal{M})$.

The binary shuffle operation is denoted $\sqcup\!\sqcup$. $\mathcal{M}_1 \sqcup\!\sqcup \mathcal{M}_2$ is again a set of split- `shuffle` CBMs as there are several possible ways of rearranging the rigid components preserving the partial order.

For $i \in \{1,2\}$ let $\mathcal{M}_i = (\mathcal{E}_i, \lambda_i, \mathsf{pid}_i, \xrightarrow{r}_i, \xrightarrow{e}_i, \rhd_i)$ such that $\mathcal{E}_1 \cap \mathcal{E}_2 = \emptyset$. Further let $\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \xrightarrow{r}, \xrightarrow{e}, \rhd)$ be a plit-CBM. $\mathcal{M} \in \mathcal{M}_1 \sqcup\!\sqcup \mathcal{M}_2$ if

- $\mathcal{E} = \mathcal{E}_1 \uplus \mathcal{E}_2$, $\lambda = \lambda_1 \uplus \lambda_2$, $\mathsf{pid} = \mathsf{pid}_1 \uplus \mathsf{pid}_2$, $\rhd = \rhd_1 \uplus \rhd_2$
- $\xrightarrow{r} = \xrightarrow{r}_1 \uplus \xrightarrow{r}_2$. The components are the disjoint union of the constituent components.



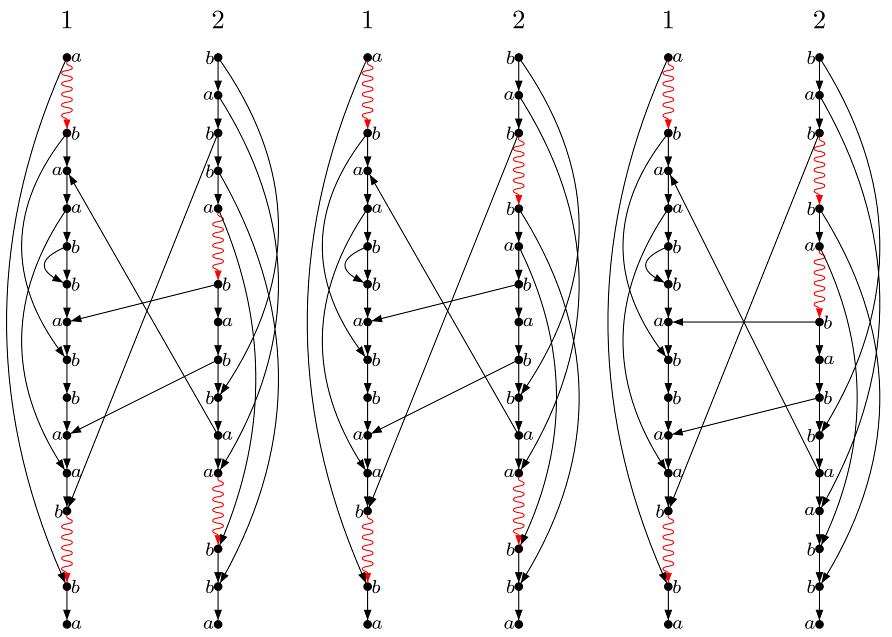


Figure 3.6: The set $\mathsf{merge}(\mathcal{A}_{12})$

- $\xrightarrow{\text{e}}_1 \cup \xrightarrow{\text{e}}_2 \subseteq (\xrightarrow{\text{r}} \cup \xrightarrow{\text{e}})^*$. The ordering dictated by the elastic edges of $\mathcal{M}_i$ is respected in the shuffle.

Remark 3.13. All the components on a process are totally ordered by the $\xrightarrow{\text{e}}$ edges. This respects the ordering given by $\xrightarrow{\text{e}}_1$ and $\xrightarrow{\text{e}}_2$. Thus any event participating in an elastic edge still does so after a shuffle. Moreover, two consecutive components in $\mathcal{M}_i$ can be separated only by components from $\mathcal{M}_{3-i}$.

More formally, for $i \in \{1, 2\}$, if $(e_i, f_i) \in \xrightarrow{\text{e}}_i$ then $\exists e, f \in \mathcal{E}$ such that $(e_i, e) \in \xrightarrow{\text{e}}$ and $(f, f_i) \in \xrightarrow{\text{e}}$. Moreover, for all $e \in \mathcal{E}$ if $e_i(\xrightarrow{\text{e}} \cup \xrightarrow{\text{r}})^+ e(\xrightarrow{\text{e}} \cup \xrightarrow{\text{r}})^+ f_i$, then $e \in \mathcal{E}_{3-i}$.

Let $\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \xrightarrow{\text{r}}, \xrightarrow{\text{e}}, \rhd)$ be a split-CBM. We denote the active processes in $\mathcal{M}$ by $\mathsf{Procs}(\mathcal{M}) \subseteq \mathbf{Procs}$. That is, $\mathsf{Procs}(\mathcal{M}) = \{p \in \mathbf{Procs} \mid$ there is an event $e \in \mathcal{E}$ such that $\mathsf{pid}(e) = p\}$.

Claim 3.14. $|\xrightarrow{\text{e}}| = |\mathsf{Procs}(\mathcal{M}_1) \cap \mathsf{Procs}(\mathcal{M}_2)| + |\xrightarrow{\text{e}}_1| + |\xrightarrow{\text{e}}_2|$.

Proof. The claim follows from the observations:

OBS1 The number of elastic edges on Process p remains the same in $\mathcal{M}_i$ and in $\mathcal{M}_i \sqcup \mathcal{M}_{3-i}$, if p is not present in $\mathcal{M}_{3-1}$: For $p \in \mathsf{Procs}(\mathcal{M}_i) \setminus \mathsf{Procs}(\mathcal{M}_{3-i})$, $|\xrightarrow{\text{e}} \cap \mathsf{pid}^{-1}(p) \times \mathsf{pid}^{-1}(p)| = |\xrightarrow{\text{e}}_i \cap \mathsf{pid}_i^{-1}(p) \times \mathsf{pid}_i^{-1}(p)|$.

OBS2 If a Process p is present in both $\mathcal{M}_1$ and $\mathcal{M}_2$, then the number of elastic edges on Process p in the shuffle is one plus the sum of the number of elastic edges on Process p in $\mathcal{M}_1$ and $\mathcal{M}_2$: For $p \in \mathsf{Procs}(\mathcal{M}_1) \cap \mathsf{Procs}(\mathcal{M}_2)$, $|\xrightarrow{\text{e}} \cap \mathsf{pid}^{-1}(p) \times \mathsf{pid}^{-1}(p)| = 1 + |\xrightarrow{\text{e}}_1 \cap \mathsf{pid}_1^{-1}(p) \times \mathsf{pid}_1^{-1}(p)| + |\xrightarrow{\text{e}}_2 \cap \mathsf{pid}_2^{-1}(p) \times \mathsf{pid}_2^{-1}(p)|$. □

Example 3.15. Consider the split-CBMs $\mathcal{M}_{141}$ and $\mathcal{M}_{142}$ given in Figure 3.7. The split-CBM $\mathcal{M}_{12}$ is in $\mathcal{M}_{141} \sqcup \mathcal{M}_{142}$.

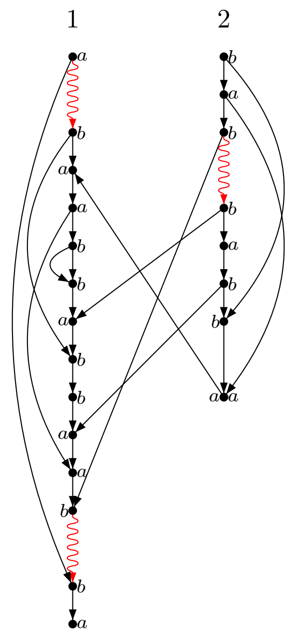
The set $\mathcal{M}_{141} \sqcup \mathcal{M}_{142}$ contains six split-CBMs in total. They are given in Figure 3.8.

Example 3.16. Consider the split-CBMs $\mathcal{M}_{151}$ and $\mathcal{M}_{152}$ given in Figure 3.9. The split-CBM $\mathcal{M}_{12}$ is in $\mathcal{M}_{151} \sqcup \mathcal{M}_{152}$. All the CBMs shown in Figure 3.8 are also in $\mathcal{M}_{151} \sqcup \mathcal{M}_{152}$. In fact the set $\mathcal{M}_{151} \sqcup \mathcal{M}_{152}$ contains 18 split-CBMs in total.

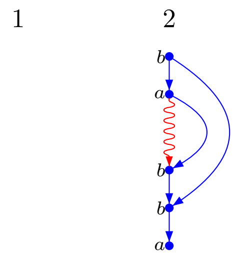
We can lift the definition of merge and shuffle to sets of split-CBMs in the natural way. Let L, L_1 and L_2 be sets of split-CBMs. We define

$$\mathsf{merge}(L) = \bigcup_{\mathcal{M} \in L} \mathsf{merge}(\mathcal{M}) \tag{3.1}$$

$$L_1 \sqcup L_2 = \bigcup_{\mathcal{M}_1 \in L_1, \mathcal{M}_2 \in L_2} \mathcal{M}_1 \sqcup \mathcal{M}_2 \tag{3.2}$$



(a) $\mathcal{M}_{141}$



(b) $\mathcal{M}_{142}$

Figure 3.7

3.5 An algebra over split-CBMs

We now define an algebra on split-CBMs. The basic terms of this algebra are 1) a CBM with a single event, and 2) a split-CBM with two events connected by a $\rhd$ edge and without any rigid edges ($|\xrightarrow{\text{r}}| = 0$).

There are two operations in this algebra: a unary merge and a binary shuffle. These operations are analogous to the merge and shuffle defined above, and hence we use the same name for them.

A *split-term* over $(\Sigma, \mathbf{Procs})$ is of the form:

$$s := (a,p) \mid (a,p) \rhd (b,p') \mid \mathsf{merge}(s) \mid s \sqcup\!\sqcup\, s$$

where $a, b \in \Sigma$ and $p, p' \in \mathbf{Procs}$.

Each split-term represents a set of split-CBMs. For each split-term s, the set of split-CBMs it represents, denoted $[\![s]\!]$, is given as follows:

- $[\![(a,p)]\!]$ is the CBM with a single node labelled a on process p.
- $[\![(a,p) \rhd (b,p')]\!]$ is the CBM with two nodes labelled a (resp. b) on process p (resp. p') connected by a $\rhd$ edge. Moreover, if $p = p'$, these two nodes are linked by an elastic edge: $(a,p) \xrightarrow{\text{e}} (b,p)$

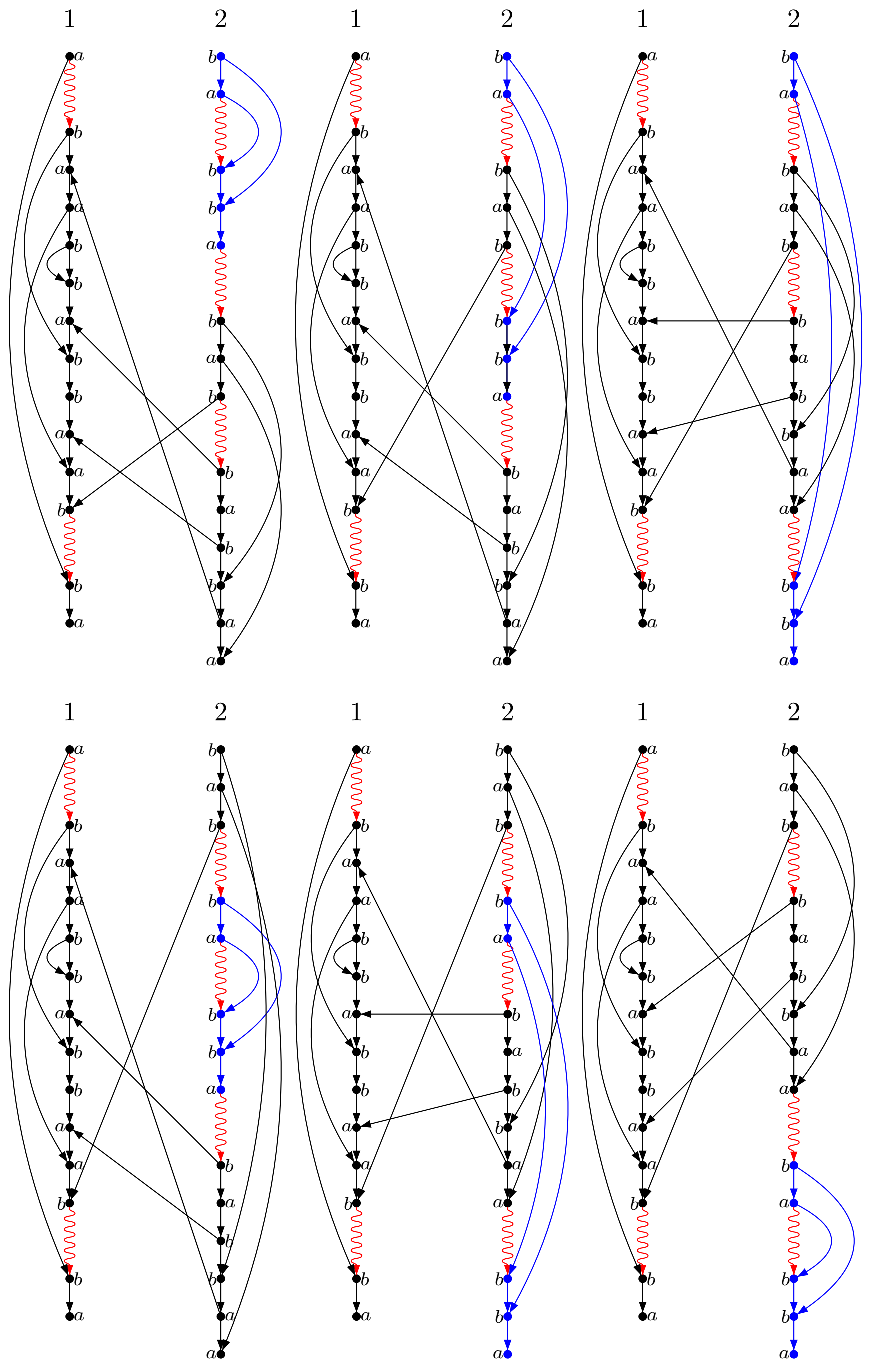
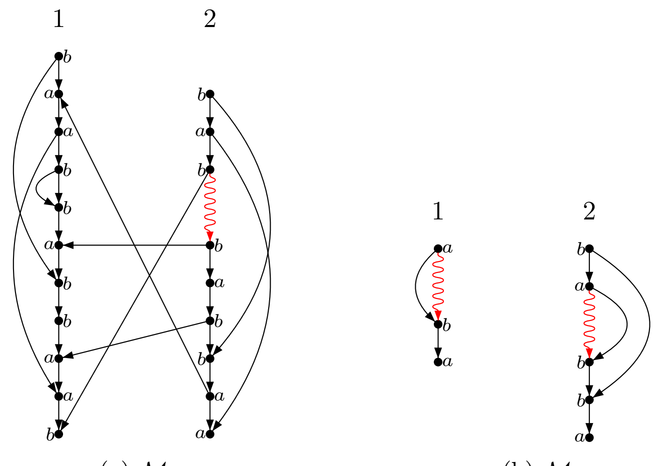


Figure 3.8: The set $\mathcal{M}_{141} \sqcup \mathcal{M}_{142}$



(a) $\mathcal{M}_{151}$ (b) $\mathcal{M}_{152}$

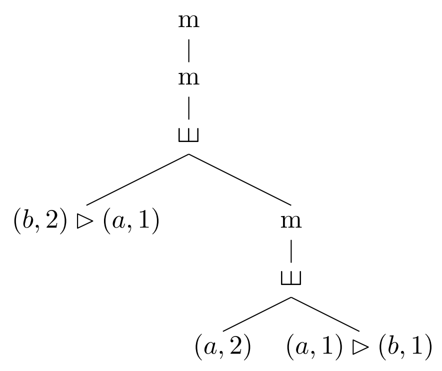
Figure 3.9

- $[\![\mathsf{merge}(s)]\!] = \mathsf{merge}([\![\mathcal{M}]\!])$ (cf. (3.1)).

- $[\![s_1 \sqcup s_2]\!] = [\![s_1]\!] \sqcup [\![s_2]\!]$ (cf. (3.2)).

We call $[\![s]\!]$ the *semantics* of the split-term s. `semantics of split-term`

Example 3.17. Consider the split-term $\mathsf{merge}(\mathsf{merge}(((b,2) \rhd (a,1)) \sqcup (\mathsf{merge}((a,2) \sqcup ((a,1) \rhd (b,1))))))$. This term can be pictorially depicted as shown on the right. The semantics of the sub-terms are depicted in Figure 3.10.



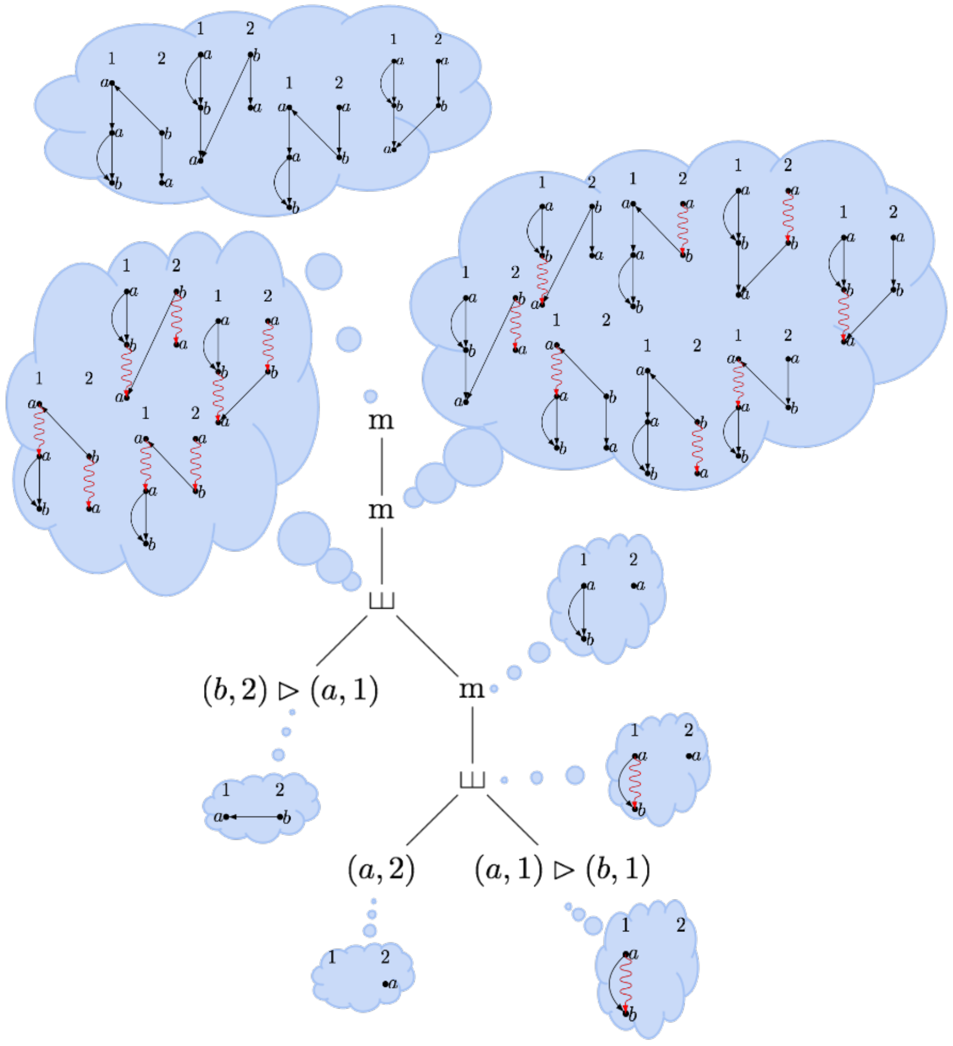
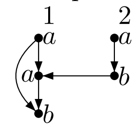
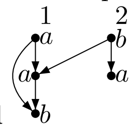


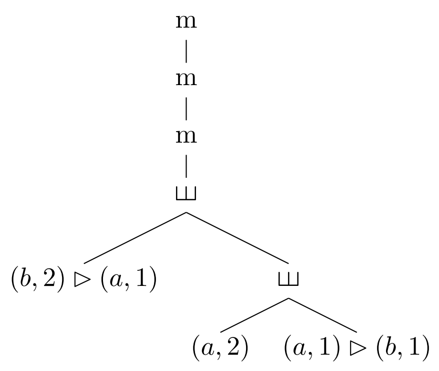
Figure 3.10: The semantics of the split-term given in Example 3.17

Example 3.18. Another split-term is shown on the right. The semantics of this split-term contains the semantics of the split-term in Example 3.17. It has



and


in its semantics, in addition to all those from Example 3.17. Notice that both terms use the same set of basic terms and the same number of merge and shuffle operations.



All the split-CBMs in $[\![s]\!]$ have the same set of non-empty processes. We define $\mathsf{Procs}(s)$ as follows:

- $\mathsf{Procs}((a,p)) = \{p\}$,
- $\mathsf{Procs}((a,p) \rhd (b,p')) = \{p,p'\}$,
- $\mathsf{Procs}(\mathsf{merge}(s)) = \mathsf{Procs}(s)$, and
- $\mathsf{Procs}(s_1 \sqcup\!\sqcup s_2) = \mathsf{Procs}(s_1) \cup \mathsf{Procs}(s_2)$

Remark 3.19. For every split-CBM $\mathcal{M} \in [\![s]\!]$, we have $\mathsf{Procs}(\mathcal{M}) = \mathsf{Procs}(s)$.

In fact all the split-CBMs defined by a split-term s have the same elasticity as well. We define the *elasticity* of a split-term s as follows.

`elasticity of split-term`

- $\mathsf{elasticity}((a,p)) = 0$,
- $\mathsf{elasticity}((a,p) \rhd (b,p')) = \begin{cases} 0 & \text{if } p \neq p' \\ 1 & \text{if } p = p' \end{cases}$,
- $\mathsf{elasticity}(\mathsf{merge}(s)) = \mathsf{elasticity}(s) - 1$, and
- $\mathsf{elasticity}(s_1 \sqcup\!\sqcup s_2) = \mathsf{elasticity}(s_1) + \mathsf{elasticity}(s_2) + |\mathsf{Procs}(s_1) \cap \mathsf{Procs}(s_2)|$

Remark 3.20. The elasticity of a split-term s is the elasticity of any (or all) split-CBM(s) in $[\![s]\!]$. That is, for every split-CBM $\mathcal{M} \in [\![s]\!]$, we have $\mathsf{elasticity}(\mathcal{M}) = \mathsf{elasticity}(s)$.

The *width* of a split-term s, denoted $\mathsf{swd}(s)$, is the maximum elasticity of all its sub-terms. The *split-width* of a split-CBM $\mathcal{M}$, denoted $\mathsf{swd}(\mathcal{M})$, is the minimum width of all split-terms s such that $\mathcal{M} \in [\![s]\!]$.

`width`
`split-width`

We say a split-term s is a k-split-term if its width is at most k.

Example 3.21. The elasticity of the split-terms in both Example 3.17 and Example 3.18 is zero. However, the split-width of the former is two, whereas that of the latter is three. Notice that the split-term with higher split-width allows more split-CBMs in its semantics though both terms use the same set of basic terms and the same number of shuffles and merges.

Remark 3.22. The split-width algebra over $(\Sigma, \mathbf{Procs})$ can generate any CBM over $(\Sigma, \mathbf{Procs})$. In fact a sequence of shuffles of basic split-terms will generate a split-CBM $\cancel{\mathcal{M}}_1 = (\mathcal{E}, \lambda, \mathsf{pid}, \xrightarrow{\mathrm{r}}_1 = \emptyset, \xrightarrow{\mathrm{e}}_1, \rhd)$. This will then be followed by a sequence of merges to get $\cancel{\mathcal{M}}_2 = (\mathcal{E}, \lambda, \mathsf{pid}, \xrightarrow{\mathrm{r}}_2, \xrightarrow{\mathrm{e}}_2 = \emptyset, \rhd)$.

Example 3.23 (Split-width of Nested-Words)**.** Nested words are MSCNs over an architecture with only one process and one stack. The split-width of any nested-word is at most two. This can be proved inductively. We give a split-term s_w for any nested-word w such that its width it at most two ($\mathsf{swd}(s_w) \leq 2$) and its elasticity is zero ($\mathsf{elasticity}(s_w) = 0$). We omit the process label and the stack label as these are unique. The base cases are:

- $w = a$ is a single event labelled a with no stack access. In this case, the split-term is simply a.

- $w = \begin{tikzcd}a \arrow[r] \arrow[r, bend left] & b\end{tikzcd}$: In this case, the split-term is $s_w = \mathsf{merge}(a \rhd b)$.

For the inductive case, we have the following case distinctions.

- If the first event does not access the stack (that is $w = a \cdot w'$): $s_w = \mathsf{merge}(a \sqcup\!\sqcup\, s_{w'})$. By induction, the split-width of $s_{w'}$ is at most 2 and its elasticity is zero. Hence, the elasticity of the topmost shuffle node of s_w is one, and hence its split-width is again at most 2. Moreover, $\mathsf{elasticity}(s_w) = 0$.

- If $w = \begin{tikzcd}a \arrow[r] \arrow[rr, bend left] & w_1 \arrow[r] & b \arrow[r] & w_2\end{tikzcd}$: Let $w' = \begin{tikzcd}a \arrow[r] \arrow[rr, bend left] & w_1 \arrow[r] & b\end{tikzcd}$. Then, $s_w = \mathsf{merge}(s_{w'} \sqcup\!\sqcup\, s_{w_2})$. By induction, the split-width of $s_{w'}$ and s_{w_2} is at most 2 and their elasticity is zero. Hence, the elasticity of the topmost shuffle node of s_w is one, and hence its split-width is again at most 2. Moreover, $\mathsf{elasticity}(s_w) = 0$.

- If $w = \begin{tikzcd}a \arrow[r] \arrow[rr, bend left] & w_1 \arrow[r] & b\end{tikzcd}$: Then, $s_w = \mathsf{merge}(\mathsf{merge}(s_{w_1} \sqcup\!\sqcup\, a \rhd b))$. By induction, the split-width of s_{w_1} is at most 2 and its elasticity is zero. Hence, the elasticity of the topmost shuffle node of s_w is two, and hence its split-width is again at most 2. Moreover, $\mathsf{elasticity}(s_w) = 0$.

Chapter 4

Parametrised Decision Problems

Contents

In this chapter, we consider the parametrised versions of the decision problems considered in Section 2.7 of Chapter 2. The parameter we employ is split-width.

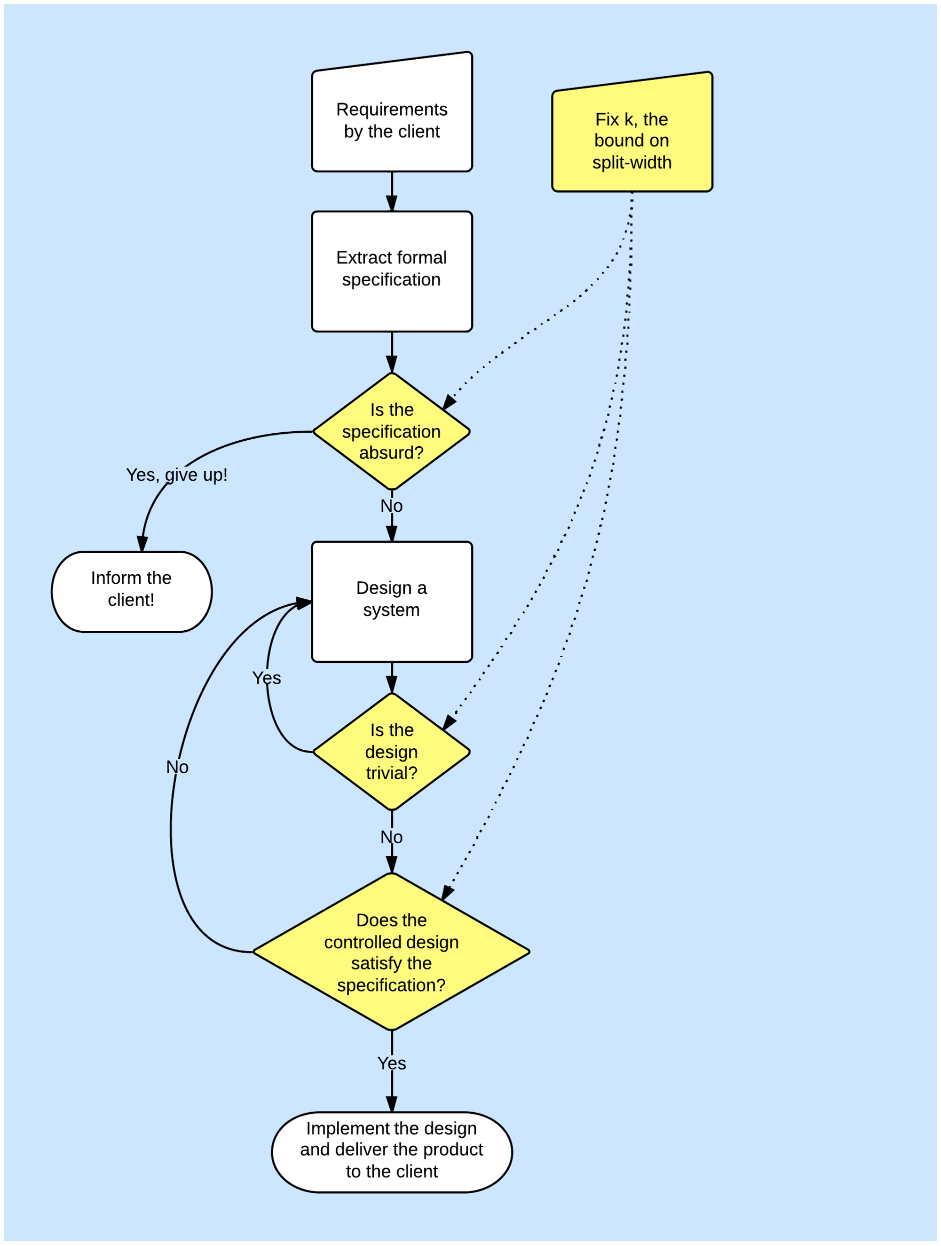

Requirements by the client
Fix k, the bound on split-width
Extract formal specification
Is the specification absurd?
Yes, give up!
Inform the client!
No
Design a system
Yes
Is the design trivial?
No
No
Does the controlled design satisfy the specification?
Yes
Implement the design and deliver the product to the client

Figure 4.1: The product development cycle with the bound on split-width as a parameter.

Figure 4.1 depicts the product development cycle of the hypothetical firm. This time they have a bound on split-width as a parameter to their aid.

The parametrised versions of these decision problems are decidable. Thus, up to the parameter, the firm can guarantee the correctness of their design. A higher value of the parameter split-width means that more behaviours have been taken into account by the decision problem. Thus, a higher value of the parameter provides a more accurate verification cycle. But, accuracy comes at the price of time and space resources. The complexity of these procedures increase with the value of the parameter.

The engineers may do the tests against different values of the parameter. If the tests pass positively with a particular value, and if they have time, they may increment the value for better accuracy. On the other hand, if it does not work for one particular value, they may decrease the value and see whether for a lower value it works. The figure demonstrates the scenario for one fixed value of split-width.

4.1 Preliminaries

Our decision procedures rely on tree-automata techniques. We are dealing with only finite, binary trees in this chapter (and in the thesis).

Here we will briefly recall some automata formalisms over binary trees and the complexities of their emptiness checking problem.

A finite binary tree T over Σ is a Σ labelled finite tree $T : \mathsf{dom}(T) \to \Sigma$ in which every node has at most two children. All nodes except the root has a parent. The parent of a node can be reached by following the up direction. Some nodes have only one child. The child of such a node node can be reached by following the left direction. Some nodes have two children, which can be accessed respectively by following the left and right directions.

The set of possible directions of a binary tree is thus $\mathsf{D} = \{\mathsf{up}, \mathsf{left}, \mathsf{right}\}$. Only a subset of these directions are available at a particular node.

Alternating 2-way Tree Automata We present here an adapted definition of alternating 2-way automata [Var98]. The orginal definition was for infinite trees and infinite runs which require a parity acceptance condition. However, our purposes need only finite runs on finite trees. Hence we remove the parity acceptance condition from the definition.

Definition 4.1. An *alternating 2-way tree automaton* (A2A) over Σ labelled binary trees is a tuple $\mathcal{A} = (Q, \delta, \mathsf{Acc})$ where A2A

- Q is a finite set of *states*,
- $\mathsf{Acc} \subseteq Q$ is the subset of *accepting* or *initial* states, and

- $\delta : Q \times \Sigma \times 2^{\mathsf{D}} \to \mathcal{B}^+(\textsc{Moves} \times Q)$ is the transition function where $\textsc{Moves} = \{\mathsf{stay}, \mathsf{up}, \mathsf{left}, \mathsf{right}\}$ and $\mathcal{B}^+(\textsc{Moves} \times Q)$ is the set of positive boolean formulas over $\textsc{Moves} \times Q$.

We use A2A occasionally, so we only give an intuition of their semantics and refer to [Var85] for details.

A *run* of an A2A $\mathcal{A}$ over a Σ-labelled finite tree T is a $(Q \times \mathsf{dom}(T))$-labelled finite tree ρ such that for each node $x \in \mathsf{dom}(\rho)$ labelled (q,u), if $x_1, \dots, x_n$ are the children of x in ρ and are labelled $(d_1, q_1), \dots, (d_n, q_n)$, then $\{(d_1, q_1), \dots, (d_n, q_n)\} \models \delta(q, \lambda(u), \Delta(u))$ where $\Delta(u)$ is the set of directions available at u. Notice that, according to the definition above, each leaf of ρ must be labelled by pairs (q,u) such that $\delta(q, \lambda(u), \Delta(u)) = \mathtt{true}$. The run ρ is *accepting* if its root is labelled (q,u) with $q \in \mathsf{Acc}$ and u being the root of T. A tree T is accepted by $\mathcal{A}$ if there is an accepting run ρ of $\mathcal{A}$ over T.

Fact 4.2 ([Var98]). *Given an A2A $\mathcal{A}$ with n states, one can check in time exponential in n if the set of trees accepted by $\mathcal{A}$ is nonempty.*

`tree-automaton` A classical *tree-automaton* is a special case of A2A which is only moving downwards and which sends exactly one thread to each children of a node. Thus, we have transitions of the form $\delta(q, a, \Delta) = \bigvee \bigwedge_{d \in \Delta \setminus \{\mathsf{up}\}} (q_d, d)$.

Fact 4.3. *The emptiness of a tree-automaton with n states can be checked in time $\mathcal{O}(n^3)$.*

A classical tree-automaton could be seen intuitively as a mechanism for tiling a tree from the root to the leaves (top-down), or from the leaves to the root (bottom-up). When viewed bottom-up, this class of tree-automata is determinisable (with an exponential blow-up in the number of states). A (complete) *deterministic bottom-up automaton* can be complemented by complementing the set of accepting states.

`deterministic bottom-up automata`

Fact 4.4 ([Var98]). *An A2A can be translated in exponential time to an equivalent (non-deterministic) tree-automaton.*

Fact 4.5. *The class of trees definable by an MSO formula over trees is precisely the same as that recognisable by a tree automaton.*

`cascade product` The *cascade product* of two tree automata is essentially a cartesian product in which the second automaton can access the states of the first automaton while making transitions.

`tree-walking automaton` A *tree-walking automaton* is another special case of an A2A where the transitions allow only disjunctive formulas. Thus $\delta : Q \times \Sigma \times 2^{\mathsf{D}} \to \bigvee(\textsc{Moves} \times Q)$. In addition, we consider a subset $Q_0 \subseteq Q$ of initial states. A run is then a sequence $(q_0, u_0), (q_1, u_1) \cdots (q_n, u_n)$ such that for all $0 \le i < n$ we have $(q_{i+1}, u_{i+1}) \models \delta(q_i, \lambda(u_i), \Delta(u_i))$. The run goes from node u_0 to node u_n and it is accepting if $q_O \in Q_0$ and $q_n \in \mathsf{Acc}$. The *semantics* $[\![\mathcal{A}]\!]_T$ of the automaton over the tree T is the set of pairs (u,v) such that there is an accepting run of $\mathcal{A}$ over T from u to v.

4.2 Towards decidability

A main result of this chapter is that, for the class of CBMs with bounded split-width, satisfiability checking of MSO and PDL formulas are decidable. It follows that various model-checking problems for systems of concurrent processes with data-structures are also decidable once a bound of split-width is assumed.

The proof of decidability is based on the following idea: Represent a (split-) CBM $\mathcal{M}$ of split-width at most k as a tree. Interpret the logics and automata over (split-) CBMs over such tree-representations. Benefit from the decidability of logics and automata over trees.

This section investigates a good tree-representation for split-CBMs with bounded split-width.

A candidate tree for such a representation is a k-split-term s such that $\mathcal{M} \in [\![s]\!]$. However, such a split-term represents several split-CBMs (all those in $[\![s]\!]$), whereas we want the tree-representations to be unique: A tree-representation should represent only one CBM.

4.2.1 Disambiguated split-terms

Let us locate where the uniqueness fails in split-terms. A basic split-terms represents a unique split-CBM. A merge introduces some ambiguity on which elastic edge is being replaced by a rigid edge, and thus compromises uniqueness. Similarly a shuffle also has several possibilities, which again spoils uniqueness.

The ambiguous merge and shuffle can be disambiguated in order to meet our purpose. This requires some additional labelling (or typing) of the operations.

For a merge, we explicitly specify which elastic edge gets transformed to a rigid one. Thus a merge node will be labelled by a pair (p, j), where $p \in \mathbf{Procs}$ and $j \in \{1, \dots, k-1\}$, meaning that the jth elastic edge on process p is replaced by a rigid edge.

Example 4.6. Consider Example 3.10. There are five split-CBMs in $\mathsf{merge}(\mathcal{M}_{12})$. Each of them can be uniquely identified by the following labelling (in the order):

$$\mathsf{merge}_{(1,1)}(\mathcal{M}_{12}),\ \mathsf{merge}_{(1,2)}(\mathcal{M}_{12}),\ \mathsf{merge}_{(2,1)}(\mathcal{M}_{12}),$$
$$\mathsf{merge}_{(2,2)}(\mathcal{M}_{12}),\ \mathsf{merge}_{(2,3)}(\mathcal{M}_{12}).$$

For a shuffle, we specify which of the components come from the first argument (left child). Thus a shuffle node is labelled by a set of pairs $S = \{(p_1, j_1), \dots\}$. If $(p, j) \in S$, then the jth component on Process p comes from the first argument. Otherwise it comes from the second argument.

Example 4.7. Consider Example 3.15. The six split-CBMs in $\mathcal{M}_{141} \sqcup\!\sqcup \mathcal{M}_{142}$

can be uniquely identified by the following labelling (in the order) :

$$\begin{gathered}
\mathcal{M}_{141} \sqcup_{\{(1,1),(1,2),(1,3),(2,3),(2,4)\}} \mathcal{M}_{142} \\
\mathcal{M}_{141} \sqcup_{\{(1,1),(1,2),(1,3),(2,2),(2,4)\}} \mathcal{M}_{142} \\
\mathcal{M}_{141} \sqcup_{\{(1,1),(1,2),(1,3),(2,2),(2,3)\}} \mathcal{M}_{142} \\
\mathcal{M}_{141} \sqcup_{\{(1,1),(1,2),(1,3),(2,1),(2,4)\}} \mathcal{M}_{142} \\
\mathcal{M}_{141} \sqcup_{\{(1,1),(1,2),(1,3),(2,1),(2,3)\}} \mathcal{M}_{142} \\
\mathcal{M}_{141} \sqcup_{\{(1,1),(1,2),(1,3),(2,1),(2,2)\}} \mathcal{M}_{142}
\end{gathered}$$

Notice that the label on a merge operation specifies an *elastic edge*, whereas that on shuffle operations specifies a set of *components*. The label on the merge node could also be seen equivalently as identifying the component which gets merged to its successive one.

We also make a minor change to the leaves of the split-terms so as to make the constructions easier. The basic split-term $(a,p) \triangleright (b,p')$ will instead be represented as a tree of size three.

$$\begin{tikzcd}
 & \triangleright \arrow[dl, dash] \arrow[dr, dash] & \\
(a,p) & & (b,p')
\end{tikzcd}$$

`k-DST` Such a *disambiguated-k-split-term* (abbr. k-DST) uniquely determines a split-CBM. A k-DST is a finitely labelled at-most-binary tree.

- The leaves are labelled by pairs (a,p) from $\Sigma \times \mathbf{Procs}$.
- The unary internal nodes are labelled by pairs (p,j) from $\mathbf{Procs}\times\{1,\ldots,k\}$.
- Some binary nodes are labelled by $\triangleright$. These nodes appear at height 1.
- The remaining binary nodes are labelled by sets of pairs of process name and component number, from $2^{\mathbf{Procs}\times\{1,\ldots,k+1\}}$.

Thus the finite labels of a k-DST comes from the set $(\Sigma \times \mathbf{Procs}) \cup \{\triangleright\} \cup (\mathbf{Procs}\times\{1,\ldots,k\}) \cup 2^{\mathbf{Procs}\times\{1,\ldots,k+1\}}$. This constitutes the finite alphabet for
`k-DST-Labels` k-DST which we denote by k-DST-Labels.

A k-DST is a binary tree over k-DST-Labels. The set of nodes of a k-DST T is denoted $\mathsf{dom}(T)$. The set of leaves of a tree T is denoted Leaves_T, and the set of leaves of the subtree of T rooted at the node $x \in \mathsf{dom}(T)$ is denoted $\mathsf{Leaves}_T(x)$.

4.2.2 Valid k-DSTs

A k-DST is valid if it actually represents a split-CBM.

Note that any binary tree over k-DST-Labels need not be a valid DST. Even if the labellings respect the typing requirements described above, it may still have invalid labels: The specified elastic edge at a merge node need not be present, or all the components specified at a shuffle node might not be present, either in the parent or in the child.

However, validity can be checked if we know how many components are present on each process at every node. We call this information (number of components per process) the 'type' of a node, and with the help of 'type' we can define valid k-DSTs.

A binary tree over k-DST-Labels is a valid k-DST if there is a mapping Type from the nodes of the tree to $\mathfrak{p}$-tuples $\{0,\dots,k\}^{\mathbf{Procs}}$, denoted $\mathsf{Type}(x) = (i_1^x, \dots i_{\mathfrak{p}}^x)$, such that the following hold for every node x: `valid` k`-DST` `Type`

DST1 If $T(x) = (a,p) \in \Sigma \times \mathbf{Procs}$, then x is a leaf, and $\mathsf{Type}(x) = (\underbrace{0,\dots,0,1}_{p}, \underbrace{0,\dots,0}_{\mathfrak{p}-p})$.

DST2 If $T(x) = \rhd$, then x has two children y and z. Both y and z must be leaves. Let $T(y) = (a,p_1)$ and $T(z) = (b,p_2)$. Then, $\mathsf{Type}(x) = \mathsf{Type}(y) + \mathsf{Type}(z)$, their component-wise sum.

For instance, if $p_1 < p_2$, $\mathsf{Type}(x) = (\underbrace{0,\dots,0,1}_{p_1}, \underbrace{0,\dots,1}_{p_2-p_1}, \underbrace{0,\dots,0}_{\mathfrak{p}-p_2}))$, and if $p_1 = p_2$, $\mathsf{Type}(x) = (\underbrace{0,\dots,0,2}_{p_1}, \underbrace{0,\dots,0}_{\mathfrak{p}-p_1})$

DST3 If $T(x) = (p,j) \in \mathbf{Procs} \times \{1,\dots,k\}$, then x has only one child, call it y, and $\mathsf{Type}(x) = (i_1^y, \dots, i_{p-1}^y, i_p^y - 1, i_{p+1}^y, \dots, i_{\mathfrak{p}}^y)$. Moreover, $j \le i_p^x$ (equivalently, $j < i_p^y$).

DST4 If $T(x) = S = \{(p_1,j_1),\dots\} \subseteq \mathbf{Procs} \times \{1,\dots,k+1\}$, then x has two children y and z. $\mathsf{Type}(x) = \mathsf{Type}(y) + \mathsf{Type}(z)$, their component-wise sum. If $(p,j) \in S$, then $j \le i_p^x$. For each p, $|\{j \mid (p,j) \in S\}| = i_p^y$.

Remark 4.8. Note that the mapping Type is unique if it exists.

A valid k-DST represents a unique split-CBM. In fact, every subtree of a k-DST T represents a unique split-CBM. Thus we can associate a split-CBM to every node $x \in \mathsf{dom}(T)$ – the one represented by the subtree rooted at x. The events of this split-CBM correspond to the leaves of the subtree rooted at x. We call it the semantics, and denoted it by $[\![x]\!]_T$. The semantics of a k-DST T is the semantics of its root. That is, $[\![T]\!] = [\![x]\!]_T$ where x is the root of T. $[\![T]\!]$

In fact, $\mathsf{Type}(x)$ is an abstraction of $[\![x]\!]_T$ since $[\![x]\!]_T$ has i_p^x components on process p.

We will now inductively define $[\![x]\!]_T = (\mathcal{E}_x = \mathsf{Leaves}_T(x), \lambda_x, \mathsf{pid}_x, \xrightarrow{\mathrm{r}}_x, \xrightarrow{\mathrm{e}}_x, \rhd_x)$, and also verify that it agrees with $\mathsf{Type}(x)$. Simultaneously, we will also verify that $[\![x]\!]_T \in [\![s_x]\!]$ where s_x is the split-term defined by the subtree of T rooted at x. $[\![x]\!]_T$

- If $T(x) = (a,p)$, then $[\![x]\!]_T$ is the CBM with a single node labelled a on process p. That is,

$$[\![x]\!]_T = (\mathcal{E}_x = \{x\}, \lambda_x = \{x \mapsto a\}, \mathsf{pid}_x = \{x \mapsto p\}, \xrightarrow{\mathrm{r}}_x = \emptyset, \xrightarrow{\mathrm{e}}_x = \emptyset, \rhd_x = \emptyset)$$

Clearly $[\![x]\!]_T$ agrees with $\mathsf{Type}(x)$, and $[\![x]\!]_T \in [\![s_x]\!]$.

- If $T(x) = \rhd$. Let the left child of x be y with $T(y) = (a, p)$ and the right child of x be z with $T(z) = (b, p')$. Then $[\![x]\!]_T$ is the unique split-CBM in $[\![(a, p) \rhd (b, p')]\!]$.

$$[\![x]\!]_T = \quad (\mathcal{E}_x = \{y, z\}, \lambda_x = \{y \mapsto a, z \mapsto b\}, \mathsf{pid}_x = \{y \mapsto p, z \mapsto p'\},$$
$$\xrightarrow{\mathrm{r}}_x = \emptyset, \xrightarrow{\mathrm{e}}_x = \begin{cases} \{(y, z)\} & \text{if } p = p' \\ \emptyset & \text{otherwise} \end{cases}, \rhd_x = \{(y, z)\})$$

  Notice that $[\![x]\!]_T$ agrees with $\mathsf{Type}(x)$, and $[\![x]\!]_T \in [\![s_x]\!]$.

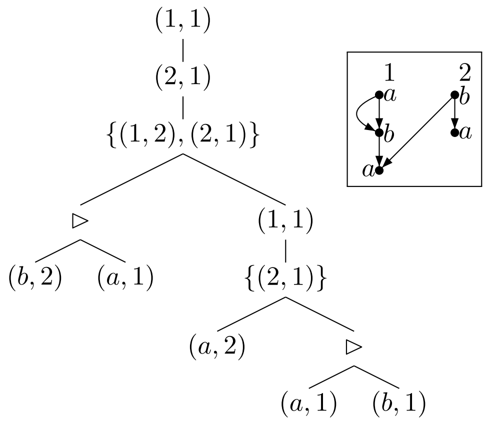
- If $T(x) = (p, j)$. Let y be the only child of x. Then $[\![x]\!]_T = \mathsf{merge}_{(p,j)}([\![y]\!]_T)$. Notice again that $\mathsf{Type}(x)$ agrees with $[\![x]\!]_T$, and $[\![x]\!]_T \in [\![s_x]\!]$.

- If $T(x) = S = \{(p_1, j_1), \ldots\}$. Let y and z be the children of x. Then

$$[\![x]\!]_T = [\![y]\!]_T \sqcup\!\sqcup_S [\![z]\!]_T$$

  Notice that $\mathsf{Type}(x)$ agrees with $[\![x]\!]$, by definition of shuffle, and $[\![x]\!]_T \in [\![s_x]\!]$. In fact, all the components specified by the set S are present in $[\![x]\!]_T$, as T is a valid k-DST (cf. condition DST4).

Example 4.9. A disambiguated version of the split-term from Example 3.17, and the CBM it represents are shown in the right. The label on the topmost shuffle node (topmost binary node) says that the second component on Process 1 and the first component on Process 2 come from the left child. The label on the topmost but one merge node (unary) says that the first elastic edge on Process 2 must be replaced by a rigid edge.



Automaton $\mathcal{A}_{k\text{-valid}}$: The set of all valid k-DSTs is recognisable. A deterministic bottom up tree automaton can remember the type of a node in its state.
$\mathcal{A}_{k\text{-valid}}$ Only consistent transitions are allowed, and the states are updated accordingly. We call this automaton $\mathcal{A}_{k\text{-valid}}$. The number of states of $\mathcal{A}_{k\text{-valid}}$ is at most $(k+1)^{\mathfrak{p}}$.

Let s be a split-term of width at most k. For each split-CBM $\mathcal{M} \in [\![s]\!]$, the split-term s can be disambiguated to a k-DST $T_{\mathcal{M}}$ such that $[\![T_{\mathcal{M}}]\!] = \mathcal{M}$. We denote by $\mathsf{disamb}(s)$ the set of all disambiguations of s. Thus $\mathsf{disamb}(s)$ is a set of valid k-DSTs.

Proposition 4.10.

$$[\![s]\!] = \{[\![x]\!]_T \mid T \in \mathsf{disamb}(s)\}$$

Thus we may conclude that valid k-DSTs are sound and complete for split-CBMs of width at most k.

Proposition 4.11. *If T is a* valid k-DST*, then $[\![T]\!]$ is a split-CBM of split-width at most k. Conversely, if $\mathcal{M}$ is a split-CBM of split-width at most k, then there is a* valid k-DST *T such that $\mathcal{M} = [\![T]\!]$.*

Proof. The soundness follows from the definition of $[\![T]\!]$. For the completeness, since $\mathcal{M}$ has split-width at most k, there is a split-term s of width at most k such that $\mathcal{M} \in [\![s]\!]$. From Proposition 4.10, there is a k-DST $T \in \mathsf{disamb}(s)$, which is a disambiguation of s, such that $\mathcal{M} = [\![T]\!]$. □

4.2.3 Retrieving a CBM from its k-DST encoding

In fact, a k-DST T offers more than just disambiguating a split-term to have a single split-CBM in its semantics. It indeed embeds the split-CBM it represents. By definition, the events of $[\![T]\!]$ are the leaves of T. Moreover, the edge relations $\rhd$, $\xrightarrow{r}$ and $\xrightarrow{e}$ can be recovered in T efficiently.

Proposition 4.12. *Let T be a valid k-DST with $[\![T]\!] = (\mathcal{E}, \lambda, \mathsf{pid}, \xrightarrow{r}, \xrightarrow{e}, \rhd)$. The relations $\xrightarrow{r}$ and $\xrightarrow{e}$ can be recovered in T*

- *by a deterministic tree-walking automaton with at most $k\mathfrak{p}$ states.*
- *by a deterministic bottom-up tree-automaton with at most $3 \times k\mathfrak{p} + 2$ states over a marked alphabet* (k-DST-Labels $\times \{0,1\}^2$).

The relation $\rhd$ can be recovered by a deterministic tree-walking automaton with at most 2 states, and equally by a deterministic bottom-up tree-automaton with at most 3 states over a marked alphabet (k-DST-Labels $\times \{0,1\}^2$).

Proof. Tree-walking automaton for $\xrightarrow{r}$: The tree-walking automaton starts at a leaf x, walks up the tree T tracking its component in its state, until it finds the first merge node labelled by its current state. Then it walks down the tree following the left-most event of merge partner (which corresponds to a leaf y). In this phase it tracks in its state the component number of the leaf y until it hits the leaf y. At every node, the update of the current state is deterministically dictated by the label of the node.

Bottom-up automaton for $\xrightarrow{r}$: We describe a bottom-up automaton $\mathcal{A}_{\xrightarrow{r}}$ with the following property: a valid k-DST T with two marked leaves x and y is accepted by $\mathcal{A}_{\xrightarrow{r}}$ (that is, $T[x,y] \in \mathscr{L}(\mathcal{A}_{\xrightarrow{r}})$) if and only if $x \xrightarrow{r} y$ in $[\![T]\!]$.

The automaton verifies that there is exactly one leaf marked x (i.e. with label $(a,p,1,0)$) and exactly one leaf marked y (i.e. with label $(b,p',0,1)$) and that these two leaves are different. A leaf x marked 1 in the second component is assigned the state $(\mathsf{pid}(x), 1, \mathsf{source})$. The state indicates that it is the first component on process $\mathsf{pid}(x)$, and the tag source says that it is assumed to be the source of the $\xrightarrow{\mathrm{r}}$ edge we are tracking for. Likewise, a leaf y marked 1 in the third component is assigned the state $(\mathsf{pid}(x), 1, \mathsf{target})$. The state with a '$\mathsf{source}$' tag tracks the component number as it moves up, while making sure that this component does not get merged on the right. Similarly, the state with a 'target' tag makes sure that this component does not get merged on the left. Once it reaches the common ancestor of the source and the target, it makes sure that the target is the successor component of source. Then it keeps track of the component of source on moving up until it sees a merge on the right when it may move into an 'accept' state. During this last leg (from common ancestor towards the merge) it ensures that no shuffle inserts a new component between the source and its successor (which is the target). This automaton needs only $3 \times k\mathfrak{p} + 2$ states: $k\mathfrak{p}$ for identifying a component number, and this could be in the source mode, target mode, or in the last leg mode. It also uses a special 'accept' state and a 'no-x-no-y' state[1]. A tree is accepted only if it has a run which assigns the accepting state to the root.

Tree-walking automaton for $\xrightarrow{\mathrm{e}}$: This is similar to that of $\xrightarrow{\mathrm{r}}$. It starts at a leaf x, walks up the tree T tracking its component in its state until the root, verifying that it does not encounter a merge node labelled by its current state. Then it walks down the tree following the left-most event of the successive component (which corresponds to a leaf y). In this phase it tracks in its state the component number of the leaf y until it hits the leaf y. At every node, the update of the current state is deterministically dictated by the label of the node.

Bottom-up automaton for $\xrightarrow{\mathrm{e}}$:This automaton is essentially $\mathcal{A}_{\xrightarrow{\mathrm{r}}}$ with a different acceptance condition. The 'accept' state of $\mathcal{A}_{\xrightarrow{\mathrm{r}}}$ is rejecting for $\mathcal{A}_{\xrightarrow{\mathrm{e}}}$, as x must not have a $\xrightarrow{\mathrm{r}}$ successor. The state corresponding to the 'last-leg' is instead accepting since it verifies that the $\xrightarrow{\mathrm{e}}$ edge is preserved until the root.

For $\rhd$ and $\rhd^{-1}$ the tree-walking automaton moves up from the leaf x to its father checking that it is a $\rhd$ node and moves down to the other child verifying that it is the leaf y. $\square$

Remark 4.13. The inverse edge relations $\xrightarrow{\mathrm{r}}^{-1}$, $\xrightarrow{\mathrm{e}}^{-1}$ and $\rhd^{-1}$ can be also recognised symmetrically. The sizes of the automata remain the same.

4.2.4 Split-MSCNs

Our aim is to reason about MSCNs rather than CBMs. The events of an MSCN are adorned with a data-structure identity (δ). The relation $\rhd$ in the case of MSCNs is the union of the matching relations of the various data-structures (that is, $\rhd = \biguplus_{d \in \mathbf{DS}} \rhd^d$). Moreover, each $\rhd^d$ has to conform to the relevent data-structure access policy – LIFO or FIFO.

In order to encode an MSCN as a CBM, we enrich the action label to include the data-structure identity as well. The requirements on $\rhd^d$ can then be checked by a tree-automaton. First we define split-MSCNs and their split-width.

[1]This state acts like an additive zero for the transitions.

Let $\mathfrak{A}$ be an architecture with the set of processes **Procs**, and data-structures **DS**. A *split-MSCN* over the architecture $\mathfrak{A}$ and a set of actions Σ is a tuple $\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \delta, \xrightarrow{\mathrm{r}}, \xrightarrow{\mathrm{e}}, \rhd)$ where `split-MSCN`

- $(\mathcal{E}, (\lambda, \delta), \mathsf{pid}, \xrightarrow{\mathrm{r}}, \xrightarrow{\mathrm{e}}, \rhd)$ is a split-CBM over $\Sigma \cup (\Sigma \times \mathbf{DS})$ and **Procs** and
- $\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \delta, \rightarrow = \xrightarrow{\mathrm{r}} \cup \xrightarrow{\mathrm{e}}, \rhd)$ is an MSCN over $\mathfrak{A}$ and Σ.

The split-width of an MSCN $\mathcal{M}$ is the split-width of its underlying CBM.

We denote the set of all MSCNs over the architecture $\mathfrak{A}$ and action set Σ with split-width at most k by $k\text{-}\mathbb{MSCN}(\mathfrak{A}, \Sigma)$. $k\text{-}\mathbb{MSCN}(\mathfrak{A}, \Sigma)$

In fact the encodings of $k\text{-}\mathbb{MSCN}(\mathfrak{A}, \Sigma)$ form a regular subset of k-DSTs over $(\Sigma \times \mathbf{DS})$ and **Procs**. We can construct an automaton which verifies that a binary tree over k-DST-Labels indeed represents an MSCN $\mathcal{M} \in k\text{-}\mathbb{MSCN}(\mathfrak{A}, \Sigma)$. For this, we need to ensure that the data-structure access policies are respected, in addition to checking that it accepts valid k-DSTs. We will describe in Proposition 4.14 the automaton $\mathcal{A}_d$ which verifies the access policy for the data-structure d. Then requisite automaton will be a product of $\mathcal{A}_{k\text{-valid}}$ and $\mathcal{A}_d$ for all $d \in \mathbf{DS}$.

Proposition 4.14. *For every data-structure $d \in \mathbf{DS}$, there is a deterministic bottom up tree-automaton $\mathcal{A}_d$ of size 2^{k^2} such that a valid k-DSTs over $(\Sigma \times \mathbf{DS})$ and* **Procs** *is accepted by $\mathcal{A}_d$ if and only if the encoded CBM respects the access policy of the data-structure d.*

Proof. The bottom up tree automaton $\mathcal{A}_d$ remembers in its state the set of pairs of components linked by a $\rhd^d$ edge. The automaton updates its state consistently at a shuffle node, as well as at a merge node. Only those states which are consistent to the access policy on d are permitted.

More formally, the states of the automaton are subsets of $\{1, \ldots, k\}^2$. If a state contains a pair (i, j) then it means that there is a $\rhd^d$ edge from an event in the ith component of $\mathsf{Writer}(d)$ to the jth component of $\mathsf{Reader}(d)$. If d is a stack, a state of the automaton must not contain two pairs (i_1, j_1) and (i_2, j_2) such that $i_1 < i_2 < j_1 < j_2$. If d is a queue, a state must not contain two pairs (i_1, j_1) and (i_2, j_2) such that $i_1 < i_2$ and $j_2 < j_1$.

Thus, those shuffles which may violate the access policy on the data-structure d are disabled. The number of states of this automaton is at most 2^{k^2}. $\square$

We denote by $\mathcal{A}_{k\text{-DS}}$ the product automaton of the various $\mathcal{A}_d$. That is $\mathcal{A}_{k\text{-DS}} = \prod_{d \in \mathbf{DS}} \mathcal{A}_d$. The deterministic bottom-up automaton $\mathcal{A}_{k\text{-DS}}$ makes sure that the access policies on all the data-structures are respected. The size of $\mathcal{A}_{k\text{-DS}}$ is $\mathcal{O}(2^{k^2 \mathfrak{d}})$. $\mathcal{A}_{k\text{-DS}}$

Finally, we obtain the deterministic bottom-up tree-automaton which recognises the encodings of $k\text{-}\mathbb{MSCN}(\mathfrak{A}, \Sigma)$ as k-DSTs over $(\Sigma \times \mathbf{DS})$ and **Procs**. We denote this automaton by $\mathcal{A}_k$, which is the product of $\mathcal{A}_{k\text{-valid}}$ and $\mathcal{A}_{k\text{-DS}}$. For acceptance, it requires in addition that the root verifies that every process has at most one component. This information (Type) is available in the states of the automaton $\mathcal{A}_{k\text{-valid}}$. The size of $\mathcal{A}_k$ is $2^{\mathcal{O}(k^2 \mathfrak{d} \mathfrak{p})}$. $\mathcal{A}_k$

As we shall see next, various verification problems on MSCNs and CPDS become decidable when parametrised by a bound on split-width. The decidability is shown via automata theoretic techniques, in fact by constructing tree automata over k-DSTs. The automaton $\mathcal{A}_k$ will be a common "module" in most of these constructions.

4.3 Decision Procedures

As alluded to before, the verification problems discussed in Chapter 2 become decidable, if parametrised with a bound on split-width. In this section we will see the parametrised versions of the verification problems and their decision procedures one by one. The decision procedures make crucial use of the notions introduced in the previous section.

4.3.1 MSO satisfiability

Problem 10 (SW-par-MSO-SAT).

Input φ: an $\mathsf{MSO}(\mathfrak{A}, \Sigma)$ formula,
$k \in \mathbb{N}$.

Question Does there exist $\mathcal{M} \in k\text{-}\mathbb{MSCN}(\mathfrak{A}, \Sigma)$ such that $\mathcal{M} \models \varphi$?

Theorem 4.15. SW-par-MSO-SAT *is decidable in time non-elementary in the size of the formula φ.*

Proof. We will construct a tree-automaton $\mathcal{A}_k(\varphi)$ over binary trees over k-DST-Labels such that it accepts k-DST encodings of k-MSCNs which model the formula φ. Recall that the automaton $\mathcal{A}_k$ recognises the encodings of k-MSCNs.

$$\mathscr{L}(\mathcal{A}_k(\varphi)) = \{T \in \mathscr{L}(\mathcal{A}_k) \mid [\![T]\!] \models \varphi\} \tag{4.1}$$

The automaton $\mathcal{A}_k(\varphi)$ is obtained as the intersection of $\mathcal{A}_k$ and $\mathcal{A}^k(\varphi)$ which is built by structural induction on φ as described below.

Recall that the events of $[\![T]\!]$ correspond to the leaves of T.

Hence, first-order and second-order variables of an MSO formula φ over MSCNs will be interpreted as first-order and second-order variables ranging over the *leaves* of k-DST T.

Therefore, in order to define the automaton $\mathcal{A}^k(\varphi)$ inductively, we consider first atomic formulas assuming that free variables are bound to (set of) leaves. Thus, to build the induction, we consider an extended alphabet where leaves are labelled with the free variables. A second order free variable may label several leaves, whereas a first order free variable labels exactly one leaf. This consistency check on first order variable labelling can be easily checked by a bottom up tree-automaton.

The cases $p(x)$ (pid test), $a(x)$ (action label test) and $d(x)$ (data-structure label test) can be verified by the k-DST-Label of the respective leaf. The cases

$x = y$ and $x \in X$ are again tests on the labels of the leaf by the extended alphabet. We can easily construct a deterministic bottom up tree automaton for these basic formulas. For the atomic binary relations $x \to y$ and $x \rhd y$, we employ the automata provided by Proposition 4.12.

Hence, we have deterministic bottom-up tree automata for all atomic formulas. The sizes of these automata are $\mathcal{O}(k\mathfrak{p})$.

The automaton for the inductive cases are obtained using the classical constructions on automata: union for disjunction, complementation for negation, and projection for existential quantification.[2]

The automaton $\mathcal{A}^k(\varphi)$ makes sure that the split-CBM encoded by any accepted DST satisfies the formula φ. The automaton $\mathcal{A}_k$ verifies that any accepted DST indeed encodes a valid MSCN in k-$\mathbb{MSCN}(\mathfrak{A}, \Sigma)$. Thus their intersection $\mathcal{A}_k(\varphi)$ indeed verifies Equation (4.1).

Theorem 4.15 follows immediately since emptiness is decidable for tree automata. □

4.3.2 PDL satisfiability

Problem 11 (SW-par-PDL-SAT).

Input	ϕ: a $\mathsf{PDL}(\mathfrak{A}, \Sigma)$ sentence, $k \in \mathbb{N}$.
Question	Does there exist $\mathcal{M} \in k$-$\mathbb{MSCN}(\mathfrak{A}, \Sigma)$ such that $\mathcal{M} \models \phi$?

Theorem 4.16. SW-par-PDL-SAT *is in* ExpTime.

Proof. If $\mathsf{E}\,\sigma$ appears in the sentence, then as before, we will construct a tree-automaton $\mathcal{A}_k(\sigma)$ over binary trees over k-DST-Labels such that it accepts k-DST encodings of k-MSCNs which model the formula σ.

$$\mathscr{L}(\mathcal{A}_k(\sigma)) = \{T \in \mathscr{L}(\mathcal{A}_k) \mid [\![\sigma]\!]_{[\![T]\!]} \neq \emptyset\}$$

The automaton $\mathcal{A}_k(\sigma)$ is actually the intersection of $\mathcal{A}_k$ and $\mathcal{A}^k(\sigma)$ where

$$\mathscr{L}(\mathcal{A}^k(\sigma)) = \{T \mid [\![\sigma]\!]_{[\![T]\!]} \neq \emptyset\}$$

We will now describe how to construct automaton $\mathcal{A}^k(\sigma)$.

In fact, we construct from a PDL formula σ an alternating two-way tree automaton (A2A) [FL79, Var85], which can then be translated to the desired tree automaton $\mathcal{A}^k(\sigma)$. The A2A is constructed inductively. We use the power of alternation to deal with the boolean connectives: non-determinism for disjunction, alternation for conjunction, and dualisation for negation. For path expression, we employ any translation from a regular expression to an automaton. This is then viewed as a walking automaton. The basic steps are simulated

[2]The projection operation makes the automaton non-deterministic, and hence it must be determinised before a complementation, and this is what makes the construction non-elementary.

by simpler walking automata described in Proposition 4.12. If a state formula need to be asserted at a node, the walking automaton invokes the corresponding alternating automaton (which is constructed by induction).

Thus the size of the A2A for σ is $\mathcal{O}(k\mathfrak{p}|\sigma|)$.

The A2A for $\mathsf{E}\,\sigma$ walks down to an arbitrary leaf and launches the A2A for σ. The boolean closure can be easily taken into account by an A2A. Thus we get an A2A for ϕ whose size is $\mathcal{O}(k\mathfrak{p}|\sigma|)$.

We can indeed obtain a bottom-up tree automaton $\mathcal{A}^k(\sigma)$ from the above A2A[Var98]. The number of states of $\mathcal{A}^k(\sigma)$ is $2^{\mathcal{O}(k^2|\sigma|^2)}$.

The PDL sentence is satisfiable with respect to k-split-width MSCNs if and only if $\mathcal{A}_k(\phi)$ is non-empty. □

`PDL with intersection`

Remark 4.17. For PDL with intersection, the satisfiability checking with respect to bounded split-width behaviours is 2ExpTime. The problem in this case is called SW-par-$\mathsf{PDL}^{\cap}$-SAT.

This bound is obtained by essentially an adaptation of the construction in [GLL09] for PDL with intersection over trees. The tree-walking alternating automaton for atomic path-expressions in their construction are very simple, which, in the adaptation, needs to be replaced with the walking automata described above. Thus the automata for basic steps have $\mathcal{O}(k\mathfrak{p})$ states. The rest of the construction remains the same. Thus it gives us a two-way alternating tree automaton with the number of states $\mathcal{O}((k|\sigma|)^{\mathrm{IW}(\sigma)})$ where $\mathrm{IW}(\sigma)$, as defined in [GLL09] is bounded from above by the total number of occurrences of the intersection operator ($\cap$) in σ.

The emptiness of an A2A can be done in exponential time. The above construction gives us an A2A whose number of states is exponential in the size of the formula σ (and hence exponential in ϕ). Hence the complexity upper bound follows.

4.3.3 Temporal Logics satisfiability

Problem 12 (SW-par-TL-SAT)**.**

Input ϕ: a $\mathsf{TL}(\mathfrak{A},\Sigma)$ sentence,
$k \in \mathbb{N}$.

Question Does there exist $\mathcal{M} \in k\text{-}\mathbb{MSCN}(\mathfrak{A},\Sigma)$such that $\mathcal{M} \models \phi$?

Theorem 4.18. SW-par-TL-SAT *is in* ExpTime.

Proof. We instead do the parametrised satisfiability check of $\mathsf{TL2PDL}(\varphi)$. □

`TL with arbitrary PDL steps`

Remark 4.19. Notice that the proof goes through verbatim if we consider the extension of the TL with arbitrary path-expressions as steps (π). Thus the parametrised satisfiability of this extended temporal logic is also decidable in ExpTime.

Remark 4.20. Consider the extension of the TL with MSO definable modalities. The parametrised satisfiability problem of this logic is decidable, thanks to Theorem 4.15. The complexity, however, is non-elementary. For a temporal logic with a fixed set of modalities, the precise complexity can be characterised to be some m-fold exponential where m depends on the alternation depth of the modality definitions. This characterisation is due to Mennicke [Men13a].

`TL with MSO definable modalities`

However, if the parameters are fixed (k, $\mathfrak{A}$ and Σ is not part of the input), then the satisfiability is in ExpTime. A decision procedure in this case is an adaptation of [GK10, BCGZ11].

4.3.4 CPDS emptiness checking

We consider the bounded split-width emptiness checking of concurrent processes with data-structures:

Problem 13 (SW-par-CPDS-Emptiness).
Input $\mathcal{S}$: a CPDS over $\mathfrak{A}$ and Σ,
$k \in \mathbb{N}$.
Question Is $\mathscr{L}(\mathcal{S}) \cap k\text{-}\mathbb{MSCN}(\mathfrak{A}, \Sigma) = \emptyset$?

Theorem 4.21. SW-par-CPDS-Emptiness *is in* ExpTime.

Proof. Theorem 4.15 and Remark 2.19 allows us to conclude the decidability of the above problem. However, we can improve the complexity by directly building an automaton $\mathcal{A}_k(\mathcal{S})$ over k-DSTs such that a DST T is accepted if and only if the encoded MSCN M_T is accepted by $\mathcal{S}$.

The automaton $\mathcal{A}_k(\mathcal{S})$ will be the *cascade product* of the automaton $\mathcal{A}_k$ and an automaton $\mathcal{A}^k(\mathcal{S})$ which we define below.

The automaton $\mathcal{A}^k(\mathcal{S})$ is intended the simulate the run of $\mathcal{S}$ on the MSCN. Notice that each rigid component of a split-MSCN is a contiguous execution trace of a process in $\mathcal{S}$. Thus a split-MSCN is a collection of several execution traces, which is closed under the data-structure accesses, meaning that, if a read event is present, then so is the corresponding write event, and vice versa. Thus $\mathcal{A}^k(\mathcal{S})$ labels each component of a split-MSCN with a pair of control locations (ℓ, ℓ') with the intended meaning that the component can potentially take the corresponding process from location ℓ to location ℓ'.

Thus we keep the set of states of $\mathcal{A}^k(\mathcal{S})$ as $\mathsf{Locs}^{2k\mathsf{p}}$. A split-term may have fewer than k components on a process. Even in this case, the automaton remembers in the state a tuple of fixed length ($2k$), and discards the last entries as “don’t-cares”. The length until which the tuple must be read is actually the number of components present at that node (cf. Type). This information is available in the state of the automaton $\mathcal{A}_{k\text{-valid}}$. Thus the automaton $\mathcal{A}^k(\mathcal{S})$ needs to access the states of the $\mathcal{A}_{k\text{-valid}}$: this is why we have considered the cascade product of the automata instead of normal intersection.

This intuition of partial system runs described above is followed in the transition rules.[3]

- A leaf (a, p) (which is not a child of $\triangleright$ node) can be labelled by (ℓ, ℓ') if $(\ell, a, \ell') \in \text{Trans}_{p:int}$
- Consider a subtree
$$\begin{tikzcd} & \triangleright \arrow[dl, no head] \arrow[dr, no head] & \\ x & & y \end{tikzcd}$$
with $T(x) = (a, p, d)$ and $T(y) = (b, p', d)$:
The leaf x can be labelled by (ℓ_1, ℓ_1') and the leaf y can be labelled by (ℓ_2, ℓ_2') and the $\triangleright$ node can be labelled by a tuple $((\ell_1, \ell_1'), (\ell_2, \ell_2'))$, if $(\ell_1, a, \ell_1', \ell) \in \text{Trans}_{p \to d}$ and $(\ell_2, \ell, b, \ell_2') \in \text{Trans}_{p' \leftarrow d}$.
- For a shuffle node, the start and end control locations of each component are inherited from the corresponding children.
- For a merge node x labelled (p, j): Let its child be y. Suppose the bottom-up tree automaton at node y assigns the pair (ℓ_1, ℓ_1') to the component (p, j) and the pair (ℓ_2, ℓ_2') to the component $(p, j + 1)$. Then the merged component (p, j) of the node x can be assigned (ℓ_1, ℓ_2') if $\ell_1' = \ell_2$. The pair of control locations for the other components of x are inherited from those of the child y.

Finally, the accepting states of the bottom-up tree-automaton $\mathcal{A}^k(\mathcal{S})$ will be tuples $((\ell_1, \ell_1'), \ldots, (\ell_\mathfrak{p}, \ell_\mathfrak{p}'))$ such that $(\ell_1, \ldots, \ell_\mathfrak{p})$ is the global initial state of the CPDS $\mathcal{S}$ and $(\ell_1', \ldots, \ell_\mathfrak{p}')$ is a global final state of $\mathcal{S}$.

Thus we get the following theorem:

Theorem 4.22. *Given a CPDS $\mathcal{S}$ and an integer k, we can construct in time exponential in k and polynomial in $\mathcal{S}$ an automaton $\mathcal{A}_k(\mathcal{S}) = \mathcal{A}_k \times \mathcal{A}^k(\mathcal{S})$ which accepts all k-DSTs T such that $\mathcal{S}$ has an accepting run on $[\![T]\!]$. The size of $\mathcal{A}_k(\mathcal{S})$ is $|\mathcal{S}|^{\mathcal{O}(k)} \times 2^{\mathcal{O}(k^2)}$.*

To answer Problem 13, we only need to check the emptiness of $\mathcal{A}_k(\mathcal{S})$. Since the emptiness of an exponential sized tree-automaton can be checked in exponential time, the bound claimed in Theorem 4.21 follows. $\square$

Consider the following problem where the bound k on split-width, as well as the architecture $\mathfrak{A}$ and the set of actions Σ is fixed a priori.

Problem 14 (SW-k-CPDS-Emptiness)**.**
Input $\mathcal{S}$: a CPDS over $\mathfrak{A}$ and Σ.
Question Is $\mathscr{L}(\mathcal{S}) \cap k\text{-}\mathbb{MSCN}(\mathfrak{A}, \Sigma) = \emptyset$?

From Theorem 4.22 it follows that

Corollary 4.23. SW-k-CPDS-Emptiness *is in* PTime.

[3]We only mention the relevant entries of the tuple here. The don't-cares could be anything.

4.3.5 Inclusion checking and Universality checking

Next we consider inclusion checking of CPDSs with respect to behaviours of bounded split-width. Given two systems $\mathcal{S}_1$ and $\mathcal{S}_2$ we would like to check whether all the MSCNs accepted by $\mathcal{S}_1$ and having split-width at most k are also accepted by $\mathcal{S}_2$.

Problem 15 (SW-par-CPDS-Inclusion)**.**

Input	$\mathcal{S}_1$, $\mathcal{S}_2$: two CPDS over $\mathfrak{A}$ and Σ,
	$k \in \mathbb{N}$.
Question	Is $\mathscr{L}(\mathcal{S}_1) \cap k\text{-}\mathbb{MSCN}(\mathfrak{A}, \Sigma) \subseteq \mathscr{L}(\mathcal{S}_2)$?

This can be done by checking whether the language of $\mathcal{A}_k(\mathcal{S}_1)$ is contained in the language of $\mathcal{A}_k(\mathcal{S}_2)$. The inclusion test for tree automata can be done by standard constructions which involves complementing the latter tree automaton.

Thus we get the following theorem.

Theorem 4.24. SW-par-CPDS-Inclusion *is in* 2ExpTime*. If the bound on split-width k is fixed,* SW-k-CPDS-Inclusion *is in* ExpTime*.*

Note that the tree-automaton $\mathcal{A}_k$ accepts all k-DSTs encoding MSCNs of split-width at most k, and the size of $\mathcal{A}_k$ is $2^{\mathcal{O}(k^2)}$. Hence, we also get the decidability of universality checking.

Problem 16 (SW-par-CPDS-Universality)**.**

Input	$\mathcal{S}$: a CPDS over $\mathfrak{A}$ and Σ,
	$k \in \mathbb{N}$.
Question	Is $k\text{-}\mathbb{MSCN}(\mathfrak{A}, \Sigma) \subseteq \mathscr{L}(\mathcal{S})$?

Theorem 4.25. SW-par-CPDS-Universality *is in* 2ExpTime*. If the bound on split-width k is fixed,* SW-k-CPDS-Universality *is in* ExpTime*.*

4.3.6 MSO model checking

Bounded split-width model checking of CPDS against MSO addresses the following problem:

Problem 17 (SW-par-MSO-MC)**.**

Input	φ: an $\mathsf{MSO}(\mathfrak{A}, \Sigma)$ formula,
	$\mathcal{S}$: a CPDS over $\mathfrak{A}$ and Σ,
	$k \in \mathbb{N}$.
Question	Do all $\mathcal{M} \in \mathscr{L}(\mathcal{S}) \cap k\text{-}\mathbb{MSCN}(\mathfrak{A}, \Sigma)$ satisfy $\mathcal{M} \models \varphi$?

This problem is decidable since it amounts to checking inclusion between tree automata, more precisely, whether the set of trees accepted by the automaton $\mathcal{A}_k(\mathcal{S})$ of Theorem 4.22 is contained in the set of trees accepted by the automaton $\mathcal{A}_k(\varphi)$ of Section 4.3.1

Theorem 4.26. SW-par-MSO-MC *is decidable in time non-elementary in the length of the* MSO *formula, exponential in k, and polynomial in the size of the CPDS.*

4.3.7 PDL and TL model checking

Problem 18 (SW-par-PDL-MC)**.**

Input ϕ: a $\mathsf{PDL}(\mathfrak{A}, \Sigma)$ sentence,
$\mathcal{S}$: a CPDS over $\mathfrak{A}$ and Σ,
$k \in \mathbb{N}$.

Question For all $\mathcal{M} \in \mathscr{L}(\mathcal{S}) \cap k\text{-}\mathbb{MSCN}(\mathfrak{A}, \Sigma)$, does $\mathcal{M} \models \phi$?

Consider the automaton $\mathcal{A}_k(\neg\phi)$ obtained in the proof of Theorem 4.16. For PDL model checking we only need to check emptiness of the intersection of $\mathcal{A}_k(\mathcal{S})$ and $\mathcal{A}_k(\neg\phi)$.

Theorem 4.27. SW-par-PDL-MC *is in* ExpTime.

Since we have a linear translation from TL to PDL (cf. Table 2.3), we get a decision procedure for TL model checking as well.

Problem 19 (SW-par-TL-MC)**.**

Input ϕ: a $\mathsf{TL}(\mathfrak{A}, \Sigma)$ sentence,
$\mathcal{S}$: a CPDS over $\mathfrak{A}$,
k: an integer.

Question For all $\mathcal{M} \in \mathscr{L}(\mathcal{S}) \cap k\text{-}\mathbb{MSCN}(\mathfrak{A}, \Sigma)$,
does $\mathcal{M} \models \varphi$?

Theorem 4.28. SW-par-TL-MC *is in* ExpTime.

4.4 Opt for Optimal

Here we summarise and optimise the complexities of our decision procedures for the various verification problems. We address the case when the parameters are part of input as well as the case when they are fixed. The parameters include not only the bound on split-width k, but also the architecture $\mathfrak{A}$ and the set of action labels Σ.

4.4.1 Optimal complexities

The complexity upper bounds of the various verification problems are summarised in Table 4.1.

These *time complexity* bounds are also optimal. For the case when the bound on split-width is not part of input follows from the known lower bounds for the simpler case of nested-words. For the case when the bound on split-width is part of input follows from the known lower bounds of the simpler case of bounded phase multi-pushdown systems[LMP07, BCGZ11]. This class of behaviours is shown to have bounded split-width ([CGN12a]; Also Section 11.2).

Consider the case of nested-words. The split-width of any nested-word is at most 2 (cf. Example 3.23). Hence the complexity upper bounds for the various decision problems follow that in the right-most column of Table 4.1.

These bounds indeed match the lower bounds shown in the literature. The lower bounds for emptiness, inclusion and universality follow from [AM09]. The lower bounds for the satisfiability and model checking of temporal logics over nested-words (NWTL) is shown in [AAB+08]. The lower bounds for PDL satisfiability and model checking is shown in [BCGZ11], where we also give the generic framework of MSO definable temporal logics capturing NWTL, and still matching the ExpTime complexity. The satisfiability and model checking against MSO is non-elementary even in the simpler case of words. Thus we conclude the lower bounds for the case when the parameter is not part of the input

The lower bounds when the parameter is part of the input follow from the case when it is fixed, except for CPDS emptiness, inclusion and universality. In fact, we can deduce a matching lower bound for CPDS emptiness checking, which we argue below in the particular case of bounded-phase multiply-nested-words. However, we only have ExpTime lower bound for inclusion and universality which we obtain from the case when the parameter is fixed.

A multiply-nested-word (MNW) is an MSCN over an architecture of multi-pushdown systems (single process and multiple stacks). A *phase* of an MNW is a contiguous sequence of events which pop from at most one stack, while no restriction on pushes is imposed. A p-phase MNW is one which can be decomposed into p phases. This restriction was introduced and studied in [LMP07], where emptiness checking of CPDS under this restriction is shown to be 2ExpTime. It is shown to be 2ExpTime-Hard in [LMP08b]. Temporal logics and PDL over this restriction was studied again in [BCGZ11], where we show that the satisfiability and model checking with respect to temporal logics and PDL are 2ExpTime, whereas they become 3ExpTime once we allow intersection in PDL. We show that a p-phase bounded MNW has split-width bounded by $\mathcal{O}(2^p)$ in [CGN12a]. We prove this bound again in a more general setting in Section 11.2. Thus, we may conclude the optimality of the decision procedure for CPDS emptiness checking as well as the asymptotic optimality of the bound on split-width of this class.

4.4.2 Further optimisations

The decision procedures uniformly make use of the encoding of MSCNs by disambiguated split-terms, and the tree-automata techniques. However, there are classes of systems over specific architectures, which indeed have bounded split-width, but the split-terms are more "linear" than tree-like. For example, the class of all words have split-width one: Any word can be canonically decomposed to have a split-term with a special structure — the right subtree of any shuffle node is simply a leaf. In fact there are other classes as well, whose decompositions have such word-like properties. We will see elaborate examples in Part III. We will now argue that we can optimise the *space complexity* in such cases by replacing the tree-automata techniques with corresponding word-automata techniques.

A split-term is *word-like* , if all the shuffle nodes have, either its left or its right, subtree with height at most a constant, say m. `word-like`

Problem	Complexity	
	When par, the bound on split-width,	
	is part of the input	is fixed
SW-par-MSO-SAT	Non-elementary	
SW-par-PDL-SAT	ExpTime-Complete	
SW-par-$\mathsf{PDL}^{\cap}$-SAT	2ExpTime -Complete	
SW-par-TL-SAT	ExpTime-Complete	
SW-par-CPDS-Emptiness	ExpTime-Complete	PTime-Complete
SW-par-CPDS-Inclusion	2ExpTime	ExpTime-Complete
SW-par-CPDS-Universality	2ExpTime	ExpTime-Complete
SW-par-MSO-MC	Non-elementary	
SW-par-PDL-MC	ExpTime-Complete	
SW-par-$\mathsf{PDL}^{\cap}$-MC	2ExpTime-Complete	
SW-par-TL-MC	ExpTime-Complete	

Table 4.1: Summary of the complexities for bounded split-width verification.

Such split-terms can be seen as a word over an extended alphabet - the label of the shuffle node contains the whole missing subtree. Thus, the extended alphabet size is $(k\text{-DST-Labels})^{2^{m+1}}$. Since m is a priori fixed constant, this amounts to a polynomial blow-up in the alphabet size.

The parametrised problems discussed in this chapter can be asked for word-like MSCNs. In this case, the complexities of the decision procedures can be improved by using finite state word automata techniques instead of tree-automata. Recall that the emptiness checking of finite state word automata is in NLogSpace where as for tree-automata, it is in PTime. Thus, this gives us a better space complexity for the decision procedures.

The complexities of the decision procedures for word-like MSCNs, in the case where the bound k on split-width is part of the input and otherwise, are summarised in Table 4.2.

Problem (Word-Like)	Complexity	
	When par, the bound on split-width,	
	is part of the input	is fixed
SW-par-MSO-SAT	Non-elementary	
SW-par-PDL-SAT	PSpace-Complete	
SW-par-PDL$^{\cap}$-SAT	ExpSpace-Complete	
SW-par-TL-SAT	PSpace-Complete	
SW-par-CPDS-Emptiness	PSpace-Complete	NLogSpace-Complete
SW-par-CPDS-Inclusion	ExpSpace-Complete	PSpace-Complete
SW-par-CPDS-Universality	ExpSpace-Complete	PSpace-Complete
SW-par-MSO-MC	Non-elementary	
SW-par-PDL-MC	PSpace-Complete	
SW-par-PDL$^{\cap}$-MC	ExpSpace-Complete	
SW-par-TL-MC	PSpace-Complete	

Table 4.2: Complexities of the decision procedures for the various verification problems for word-like MSCNs.

Remark 4.29. The notion of word-like MSCNs can be generalised to allow subtrees whose size is bounded polynomially in the split-width, rather than by a constant m. A polynomial sized subtree can be abstracted in a node by an extended labelling. The extended alphabet size in this case is again exponential in the bound on the split-width. As our k-DST-Labels was exponential in k, the extended alphabet is still polynomial in the original alphabet size. Hence even in this general case of word-like, the complexity bounds shown in Table 4.2 follow.

Remark 4.30. The lower bounds in this case follow from the known lower bounds in the case of words. The lower bounds of emptiness, inclusion and universality of CPDS follow from the case of undirected acyclic communicating finite state processes[LMP08a]. The behaviours of such systems have split-width bounded by one plus the number of processes $(1+\mathfrak{p})$. We show this bound in Section 9.2.

4.4.3 Parametrised verification with respect to parameters other than split-width

We can consider various other (parametrised) classes of MSCNs for under-approximate verification. For example, bounded phase multiply nested words with the bound on the number of phases as a parameter. The various verification problems we considered in this chapter become decidable if the class under consideration has a bounded split-width and is MSO definable.

Let $U(p)$ be a class of MSCNs parametrised by a parameter p. The parametrised verification problems and the fixed parameter verification problems can be asked in the case of the class $U(p)$ in the natural way.

Assume MSCNs in $U(p)$ have split-width at most $k = f(p)$. Further suppose k-DST encodings of MSCNs in $U(p)$ can be recognised by a tree-automaton $\mathcal{A}_{U(p)}$ of size at most $2^{\text{poly}(k)}$. Then the parametrised verification problems with respect to the class $U(p)$ are decidable. The decision procedures can be adapted by replacing the automaton $\mathcal{A}_k$ with the intersection of $\mathcal{A}_k$ and $\mathcal{A}_{U(p)}$.

The complexity upper bounds when the function f is polynomial is given in Table 4.3 and when f is exponential ($k = 2^{\text{poly}(p)}$) are given in Table 4.5. Table 4.4 and Table 4.6 summarise the complexities when the class $U(p)$ in addition admits word-like split-terms.

We will see various parametrised classes of MSCNs in Part III, where the decision procedures may reduce to the above case.

4.5 Closure Properties

Closure under *union*, *intersection* and *renaming* of CPDS holds even when restricted to bounded split-width behaviours. In fact the same constructions described in Section 2.4.3 work, preserving the same bound on split-width.

However, for *concatenation*, the bound of split-width k is preserved only if $k > \mathfrak{p}$, the number of processes. If $\mathcal{M}_1$ and $\mathcal{M}_2$ have split-width at most k for some $k > \mathfrak{p}$, then their concatenation $\mathcal{M}_1 \cdot \mathcal{M}_2$ also has split-width at most k. We explain this below.

If s_1 and s_2 are split-terms with width at most k such that $\mathcal{M}_1 \in [\![s_1]\!]$ and $\mathcal{M}_2 \in [\![s_2]\!]$, then consider $s = s_1 \sqcup s_2$. The elasticity

Problem	Complexity	
	When the parameter p	
	is part of the input	is fixed
U-par-MSO-SAT	Non-elementary	
U-par-PDL-SAT	ExpTime	
U-par-$\mathsf{PDL}^{\cap}$-SAT	2ExpTime	
U-par-TL-SAT	ExpTime	
U-par-CPDS-Emptiness	ExpTime	PTime
U-par-CPDS-Inclusion	2ExpTime	ExpTime
U-par-CPDS-Universality	2ExpTime	ExpTime
U-par-MSO-MC	Non-elementary	
U-par-PDL-MC	ExpTime	
U-par-$\mathsf{PDL}^{\cap}$-MC	2ExpTime	
U-par-TL-MC	ExpTime	

Table 4.3: Summary of the complexities of the parametrised decision procedures for the class $U(p)$ when the split-width of $U(p)$ is bounded by $k = \text{poly}(p)$, and encodings of $U(p)$ is recognisable by a $2^{\text{poly}(k)}$ tree-automaton over k-DSTs.

Problem (Word-Like)	Complexity	
	When the parameter p	
	is part of the input	is fixed
U-par-MSO-SAT	Non-elementary	
U-par-PDL-SAT	PSpace	
U-par-PDL$^{\cap}$-SAT	ExpSpace	
U-par-TL-SAT	PSpace	
U-par-CPDS-Emptiness	PSpace	NLogSpace
U-par-CPDS-Inclusion	ExpSpace	PSpace
U-par-CPDS-Universality	ExpSpace	PSpace
U-par-MSO-MC	Non-elementary	
U-par-PDL-MC	PSpace	
U-par-PDL$^{\cap}$-MC	ExpSpace	
U-par-TL-MC	PSpace	

Table 4.4: Complexities of the decision procedures for the various verification problems for the class $U(p)$ if it is admits a word-like split-term of width at most $k = \text{poly}(p)$, and when the encodings of $U(p)$ is recognisable by a tree-automaton of size at most $2^{\text{poly}(k)}$.

of s is at most $\mathfrak{p}$. Thus the split-width of s is at most $\max(k, \mathfrak{p})$.
$\mathcal{M}_1 \cdot \mathcal{M}_2 \in [\![\mathsf{merge}(\mathsf{merge}(\ldots\mathsf{merge}(s)\ldots))]\!]$.

An interesting problem which we leave open is closure under complementation with respect to bounded split-width. Though we can complement the tree-automata for the k-DST encodings of MSCNs, we do not know how to obtain a corresponding CPDS given a tree-automaton over k-DSTs.

Problem	Complexity	
	When the parameter p	
	is part of the input	is fixed
U-par-MSO-SAT	Non-elementary	
U-par-PDL-SAT	2ExpTime	ExpTime
U-par-PDL$^{\cap}$-SAT	3ExpTime	2ExpTime
U-par-TL-SAT	2ExpTime	ExpTime
U-par-CPDS-Emptiness	2ExpTime	PTime
U-par-CPDS-Inclusion	3ExpTime	ExpTime
U-par-CPDS-Universality	3ExpTime	ExpTime
U-par-MSO-MC	Non-elementary	
U-par-PDL-MC	2ExpTime	ExpTime
U-par-PDL$^{\cap}$-MC	3ExpTime	2ExpTime
U-par-TL-MC	2ExpTime	ExpTime

Table 4.5: Summary of the complexities of the parametrised decision procedures for the class $U(p)$ when the split-width k of $U(p)$ is at most exponential in p, and encodings of $U(p)$ is recognisable by a tree-automaton of size at most $2^{\text{poly}(k)}$ over k-DSTs.

Problem (Word-Like)	Complexity	
	When the parameter p	
	is part of the input	is fixed
U-par-MSO-SAT	Non-elementary	
U-par-PDL-SAT	ExpSpace	PSpace
U-par-$\mathsf{PDL}^{\cap}$-SAT	2ExpSpace	ExpSpace
U-par-TL-SAT	ExpSpace	PSpace
U-par-CPDS-Emptiness	ExpSpace	NLogSpace
U-par-CPDS-Inclusion	2ExpSpace	PSpace
U-par-CPDS-Universality	2ExpSpace	PSpace
U-par-MSO-MC	Non-elementary	
U-par-PDL-MC	ExpSpace	PSpace
U-par-$\mathsf{PDL}^{\cap}$-MC	2ExpSpace	ExpSpace
U-par-TL-MC	ExpSpace	PSpace

Table 4.6: Complexities of the decision procedures for the various verification problems for the class $U(p)$ if it is admits a word-like split-term of width k which is at most exponential in p, and when the encodings of $U(p)$ is recognisable by a tree-automaton of size at most $2^{\text{poly}(k)}$.

Chapter 5

Split-width equivalent to clique-width

Contents

In this section, we relate the notion of split-width to other well-known measures on graphs such as clique-width and tree-width. Our main result states that a bound on one of these parameters imposes a bound on the others too. This is stated in Theorem 5.1 and is depicted in Figure 5.1. This gives us a very interesting corollary: *Bounded split-width characterises MSO decidability for CBMs.*

We will now reason briefly how to conclude the corollary from Theorem 5.1. The rest of the chapter will then be mainly devoted to proving Theorem 5.1.

Theorem 5.1. *1. If the split-width of a CBM is k, then its clique-width is at most $2(k + \mathfrak{p}) + 1$.*

2. If the clique-width of a CBM is c, then its split-width is at most $2c - 3$.

For bounded degree graphs, clique-width is bounded if and only if its tree-width is bounded [CE12]. The translation between clique-width and tree-width gives us the following corollary:

Corollary 5.2. *1. If the split-width of a CBM is k, then its tree-width is at most $2(k + \mathfrak{p}) - 1$.*

2. If the tree-width of a CBM is t, then its split-width is at most $120(t + 1)$.

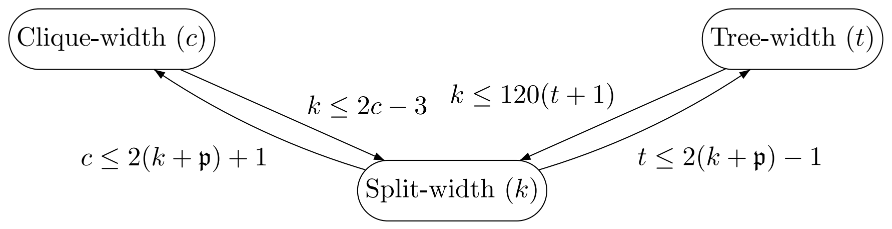


Figure 5.1: Relation between split-width, clique-width and tree-width for CBMs

Let L be a class of graphs over the vocabulary V. We say that the class L has a *decidable MSO-theory* if the following problem is decidable.

Problem 20. Input: φ: An MSO formula over the vocabulary V.
Question: Is there a graph $G \in L$ such that $G \models \varphi$?

A class of graphs L over a vocabulary V is *MSO definable*, if there exists an $\mathsf{MSO}(V)$ formula ϕ such that for all graphs G over the vocabulary V, $G \in L \iff G \models \phi$.

Fact 5.3 ([CE95])**.** *If a class of graphs is definable in MSO and has bounded clique-width, then it has a decidable MSO theory.*

The converse of Fact 5.3 is not known to be true. D. Seese conjectured that the converse is true [See91]. Courcelle [Cou06] showed that Seese's conjecture holds true for bounded degree graphs:

Fact 5.4 ([Cou06])**.** *If a class of bounded degree graphs has a decidable MSO theory, then it has bounded clique-width.*

Note that CBMs are bounded degree graphs. Thus, Fact 5.4 together with Theorem 5.1 give us a very interesting corollary:

Corollary 5.5. *Let L be a class of CBMs. If L has a decidable MSO theory, then L has bounded split-width.*

The rest of this chapter is devoted to proving Theorem 5.1.

5.1 Preliminaries

Here we will recall the definitions of clique-width and tree-width. We will also see another notion called special-tree-width, which is actually what we use in our translations.

Clique-width

Here we recall the definition of clique-width for directed graphs with node and edge labels.

Let $\mathsf{NLabels}$ be a finite set of node labels and $\mathsf{ELabels}$ be a finite set of edge relation names. A directed node labelled graph with several edge relations is a tuple $G = (V, \mathsf{N\text{-}label}, (\mathsf{E}_\gamma)_{\gamma\in\mathsf{ELabels}})$ where V is the set of vertices, $\mathsf{N\text{-}label} : V \to \mathsf{NLabels}$ is the node labelling function, and $\mathsf{E}_\gamma \subseteq V \times V$ is the edge relation γ. Such a graph can be equivalently seen as directed graph with node and edge labels, that is as a tuple $G = (V, \mathsf{N\text{-}label}, \mathsf{E\text{-}label})$ where V is the set of vertices, $\mathsf{N\text{-}label} : V \to \mathsf{NLabels}$ is the node labelling function, and $\mathsf{E\text{-}label} : V \times V \to 2^{\mathsf{ELabels}}$ gives the set of edge relations between every pair of vertices.

Definition 5.6 (k-clique-term)**.** Let $C = \{1, \ldots, k\}$ be a set of colors. The k-clique-terms over $(\mathsf{NLabels}, \mathsf{ELabels})$ are given by

$$t \ ::= \ (a, x) \ \mid \ t \oplus t \ \mid \ \mathtt{Ren}_{x,y}(t) \ \mid \ \mathtt{Add}_{x\gamma y}(t)$$

where $a \in \mathsf{NLabels}$, $\gamma \in \mathsf{ELabels}$, and $x, y \in C$. The operator $\mathtt{Ren}_,$ allows renaming of colours, and $\mathtt{Add}_\gamma$ adds a γ-edge.

Each k-clique-term t defines a graph $G^t = (V^t, \mathsf{N\text{-}label}^t, \mathsf{E\text{-}label}^t)$ and a surjective colouring relation $\chi^t \subseteq C \times V^t$.

- The term $t = (a, x)$ denotes the graph which has a single vertex v labelled a, and no edges. The colouring relation is $\chi^t = \{(x, v)\}$.
- The term $t = t_1 \oplus t_2$ denotes the disjoint union of the graphs G^{t_1} and G^{t_2}. The colouring relation is also the disjoint union: $\chi^t = \chi^{t_1} \uplus \chi^{t_2}$.
- The term $t = \mathtt{Ren}_{x,y}(t_1)$ denotes the same graph G^{t_1}. But the colouring relation recolours to y all the vertices coloured x in G^{t_1}. That is, $G^t = G^{t_1}$ and $\chi^t = \{(c, v) \mid c \neq x \text{ and } (c, v) \in \chi^{t_1}, \text{ or } c = y \text{ and } (x, v) \in \chi^{t_1}\}$.
- Finally, the term $t = \mathtt{Add}_{x\gamma y}(t_1)$ adds a γ-edge from every vertex coloured x to every vertex coloured y in G^{t_1}.

$$\mathsf{E\text{-}label}^t(v_1, v_2) = \mathsf{E\text{-}label}^{t_1} \cup \begin{cases} \{\gamma\} & \text{if } v_1 \in \chi^{t_1}(x) \text{ and } v_2 \in \chi^{t_1}(y) \\ \emptyset & \text{otherwise} \end{cases}$$

The *clique-width* of a graph G is the smallest k such that $G = G^t$ for some k-clique-term t.

Special-Tree-width [Cou10]

Special-tree-terms are clique-terms which satisfy some additional conditions. These terms use a special additional colour 0 which never gets recoloured, nor takes part in edge additions. Moreover, every other colour can colour at most one node.

Definition 5.7 (k-special-tree-term)**.** A k-special-tree-term t is a clique-term on a set of $k + 1$ colours $C = \{0, 1, 2, \ldots k\}$ such that the following conditions hold:

STW1 For every subterm t_1 of t, χ^{t_1} defines a partial function on the set $\{1, \dots, k\}$. That is, if $(x, v_1), (x, v_2) \in \chi^{t_1}$, then $v_1 = v_2$ or $x = 0$.

STW2 The color 0 is never recolored. That is, for every subterm $\mathtt{Ren}_{x,y}(t_1)$ occurring in t, we have $x \neq 0$.

STW3 The color 0 does not take part in edge additions. That is, for every subterm $\mathtt{Add}_{x\gamma y}(t_1)$ occurring in t, we have $x \neq 0$ and $y \neq 0$.

STW4 A vertex must not be created with colour 0. That is, t must not have $(a, 0)$ as a subexpression for $a \in \mathsf{NLabels}$.

The *special-tree-width* of a graph G is the smallest number k such that $G = G^t$ for some $(k+1)$-special-tree-term t.

Tree-width

The concept of *tree-width* of a graph is one of the most important measures of the algorithmic complexity of a graph and has played a fundamental role in the recent developments in graph theory. The tree-width of a graph is an integer which, as the folklore phrase goes, "tells how close a graph is to a tree" (the tree-width of a graph is 1 if and only if it is a forest).

A bound on tree-width is often shown by a "tree-decomposition" of a graph. The *tree-decomposition* of a graph is a tree whose nodes are bags of vertices of the original graph such that

1. Every vertex of the graph must be present in at least one bag.
2. If two vertices are linked by an edge in the graph, then there must be at least one bag which contains both these vertices.
3. If a vertex is contained in two bags, then it is contained in every other bag on the unique path between them in the tree-decomposition.

The *width* of a tree-decomposition is the maximum bag size minus one. The *tree-width* of a graph G is the smallest k such that G has a tree-decomposition of width k.

Which width is wider?

Note that a k-special-tree-term is a $(k+1)$-clique-term. Thus a bound on special-tree-width implies a bound on clique-width, but the converse is not true.

A k-special-tree-term also suggests a tree-decomposition of width $k-1$ which is isomorphic to the term: The bag at a sub-term t is exactly those vertices which are not coloured by 0 (i.e. $\chi^t(\{1, \dots, k\})$).

Fact 5.8. *1. The tree-width of a graph is at most its special-tree-width.*

2. The clique-width of a graph is at most its special-tree-width plus two.

In fact, a bound on tree-width implies a bound on clique-width:

Fact 5.9. *[CO00, CR01] The clique-width of any graph with tree-width k is bounded by $3 \cdot 2^{k-1}$.*

Fact 5.10. *[CO00, CR01, Cou10] For any class of graphs L,*

$$\text{special-tree-width}(L) < \infty \Longrightarrow \text{tree-width}(L) < \infty \Longrightarrow \text{clique-width}(L) < \infty,$$

but the converse implications do not hold.

However for bounded degree graphs these three notions are “equivalent”. A bound on one implies a bound on the others too.

Fact 5.11. *[CE12] For any class of bounded-degree graphs L,*

$$\text{special-tree-width}(L) < \infty \Longleftrightarrow \text{tree-width}(L) < \infty \Longleftrightarrow \text{clique-width}(L) < \infty$$

5.2 Clique-width and special-tree-width for CBMs

To generate CBMs over $(\Sigma, \mathbf{Procs})$, we set the node labels $\mathsf{NLabels} = \Sigma \times \mathbf{Procs}$ and the edge labels $\mathsf{ELabels} = \{\rightarrow, \rhd\}$.

Thus a k-clique-term for CBMs is given by:

$$t \ ::= \ (a, p, x) \ \mid \ t \oplus t \ \mid \ \mathtt{Ren}_{x,y}(t) \ \mid \ \mathtt{Add}_{x \rightarrow y}(t) \ \mid \ \mathtt{Add}_{x \rhd y}(t)$$

where $a \in \Sigma$, $p \in \mathbf{Procs}$, and $x, y \in C$.

Remark 5.12. Note that these terms can generate graphs which are not split-CBMs. Indeed we require that $\rightarrow$ edges must be added only between nodes with the same pid. We also require that a node should have at most one $\rightarrow$ predecessor and successor. These conditions need not be satisfied by arbitrary k-clique-terms. However these are regular properties on k-clique-terms, and can be ensured by a regular tree automaton.

Fact 5.13 ([Cou10, CE12]). *If a CBM has clique-width k then its special-tree-width is at most $k - 2$.*

For general graphs clique-width (c), tree-width (t) and special tree-width (s) are related as shown on the right.

$$\begin{tikzcd}
\boxed{c} & & \boxed{t} \arrow[ll, bend right, "c \leq 3 \cdot 2^{t-1}_{\text{(CO00, CR01)}}"'] \\
& \boxed{s} \arrow[ul, bend left, "c \leq s + 2_{\text{(Cou10)}}"] \arrow[ur, bend right, "t \leq s_{\text{(Cou10)}}"'] &
\end{tikzcd}$$

For CBMs as they are of degree at most three, clique-width (c), tree-width (t) and special tree-width (s) are all equivalent.

$$\begin{tikzcd}
c \arrow[rr, bend right, "t \leq c-2 \text{ (via STW)}"'] \arrow[dr, "s \leq c-2 \text{(Fact 5.13)}"'] & & t \arrow[ll, bend right, "c \leq 60t+62 \text{ (via STW)}"'] \arrow[dl, "s \leq 60(t+1) \text{(Cou10)}"] \\
& s &
\end{tikzcd}$$

5.3 Bounded split-width implies bounded clique-width

A bound on split-width implies a bound on clique-width. This follows from Fact 5.4 and Theorem 4.15, but this does not yield a bound.

In fact, we can rewrite the split-width algebra in terms of special tree-width algebra thus giving a bound on the special tree-width (and hence clique-width and tree-width).

Theorem 5.14. *If a CBM over* $(\Sigma, \mathbf{Procs})$ *has split-width at most* k*, then its special-tree-width is at most* $2(k+\mathfrak{p})-1$

Proof. Let $\mathcal{M}$ be a CBM with split-width at most k. We consider a (disambiguated) k-split-term s for $\mathcal{M}$. Each sub-term $s'$of s represents a unique subgraph of $[\![s]\!]$, which is a split-CBM given by $[\![s']\!]$. The idea is to label the end-points of the components of $[\![s']\!]$ with unique colours. The number of such end points is at most $2(k+\mathfrak{p})$. We label the end points of the components using colours from $\{1,\ldots,2(k+\mathfrak{p})\}$. The remaining nodes will be coloured 0.

The special-tree-term corresponding to a disambiguated split-term s is denoted $\langle s\rangle$. We define it inductively as follows. Intuitively, $G^{\langle s\rangle}$ gives the rigid part of $[\![s]\!]$.

- If $s=(a,p)$ then $\langle s\rangle=(a,p,x)$ for some $x\in\{1,\ldots,k\}$.
- If $s=(a,p)\rhd(b,p')$ then $\langle s\rangle=\mathtt{Add}_{x\rhd y}((a,p,x)\oplus(b,p',y))$.
- If $s=s_1\sqcup_S s_2$ then $\langle s\rangle=\langle s_1\rangle\oplus\langle s_2\rangle$.
- If $s=\mathtt{merge}_{(p,j)}(s_1)$ then let $t'=\mathtt{Add}_{x\to y}(\langle s_1\rangle)$ where x,y are such that the merge joined $\chi^{t_1}(x)$ and $\chi^{t_1}(y)$.

  $$\text{Let } t''=\begin{cases}\mathtt{Ren}_{x,0}(t') & \text{if } \chi^{t'}(x) \text{ is not the first event of any component in } G^{t'} \text{ (or } [\![s]\!])\\ t' & \text{otherwise.}\end{cases}$$

  $$\text{Then } \langle s\rangle=\begin{cases}\mathtt{Ren}_{y,0}(t'') & \text{if } \chi^{t''}(y) \text{is not the last event of any component in } G^{t''} \text{ (or } [\![s]\!])\\ t'' & \text{otherwise.}\end{cases}$$

□

5.4 Bounded clique-width implies bounded split-width

In fact our simple split-width algebra is powerful enough to capture all CBMs with bounded clique-width.

Theorem 5.15. *Every CBM of clique-width at most $k+1$ has split-width at most $2k-1$.*

This is the more involved direction. Our proof takes advantage of Fact 5.13 and shows a translation from a k-special-tree-term to a split-term of width $2k-1$.

We consider two normal forms for special-tree-terms. A special tree-term is $\rhd$-rename-free if a node taking part in a $\rhd$ edge addition is never recoloured between its creation and this edge addition: if $\texttt{Add}_{x\rhd y}(t')$ is a subterm, then the nodes taking part in this edge addition are not recolored in t'. $\rhd$-rename-free

The other normal form we consider is called $\rhd$-leafed. In a $\rhd$-leafed special-tree-term, the addition of $\rhd$ edges takes place "almost" at the leaf-level (at height one, to be precise). If a $\rhd$-leafed special-tree-term has a subterm $\texttt{Add}_{x\rhd y}(t)$ then, $t = (a,x) \oplus (b,y)$. $\rhd$-leafed

Notice that a $\rhd$-leafed term is also $\rhd$-rename-free.

The proof of Theorem 5.15 is done in several steps. First we show that any special-tree-term can be written equivalently as a $\rhd$-rename-free term (Lemma 5.16). Then we show that a $\rhd$-rename-free term can be written equivalently as a $\rhd$-leafed term (Lemma 5.18). Finally we give a translation from $\rhd$-leafed-terms to split-terms (Lemma 5.20).

Lemma 5.16. *For every k-special-tree-term t, there is an equivalent k-special-tree-term t' which is* $\rhd$-rename-free.

Proof. It is a consequence of the following even stronger lemma. □

Lemma 5.17. *For every k-special-tree-term t, there is an equivalent k-special-tree-term t' such that if a sub-term $\texttt{Ren}_{x,y}(t'')$ appears in t', then $y = 0$.*

Proof. Suppose the term t contains a subterm $\texttt{Ren}_{x,y}(t_1)$ with $y \neq 0$. We can assume that there is exactly one node coloured $x \neq 0$ and no node coloured $y \neq 0$, as t satisfies the conditions in Definition 5.7.

To rewrite in the normal form, we swap the colours x and y in the sub-term t_1 and ignore the recolouring. More formally, let π be a permutation mapping x to y and vice versa, and which is identity mapping on the other colours. We replace each occurrence of the sub-term $\texttt{Ren}_{x,y}(t_1)$ in t with $\pi(t_1)$ to get the term t''. Note that the term we obtain so is still a special-tree-term, but with fewer $\texttt{Ren}_{x,y}(\cdot)$ operations. We repeat the process until all $\texttt{Ren}_{x,y}(\cdot)$ with $y \neq 0$ are removed, and this gives us the desired term t'. □

Lemma 5.18. *For every k-special-tree-term t which is* $\rhd$-rename-free*, there is an equivalent $2k$-special-tree-term t' which is* $\rhd$-leafed.

Proof. The translation to a $\triangleright$-leafed-term is achieved at the expense of additional colours. We may use up to k additional colours from $\{1', \ldots, k'\}$.

We will bring the $\texttt{Add}_{x \triangleright y}$ terms to leaf level one by one, starting from innermost to outermost. The resulting term at each step has one less $\texttt{Add}_{x \triangleright y}$ term which is not at leaf level. Moreover this term satisfies a stronger version of $\triangleright$-rename-free. The stronger version of $\triangleright$-rename-free takes the new colours $\{1', \ldots, k'\}$ into account.

`strongly-`$\triangleright$`-rename-free` A special tree-term is *strongly*-$\triangleright$-rename-free if for every sub-term $\texttt{Add}_{x \triangleright y}(t')$ where t' is not of the form $(a,x) \oplus (b,y)$, we have the following (letting $u = \chi^{t'}(x)$ and $v = \chi^{t'}(y)$)

SRF1 $x, y \in \{1, \ldots, k\}$

SRF2 Vertices u and v are never recoloured in the sub-term t': if t'' is a sub term of t' then $u \in V^{t''} \implies \chi^{t''}(x) = u$ and $v \in V^{t''} \implies \chi^{t''}(y) = v$.

SRF3 The colour x' never appears in the branch of u and the colour y' never appears in the branch of v: if t'' is a sub-term of t' then $u \in V^{t''} \implies \chi^{t''}(x')$ is undefined and $v \in V^{t''} \implies \chi^{t''}(y')$ is undefined.

First we will see how to bring an innermost $\texttt{Add}_{x \triangleright y}$ to the leaf level. This case is depicted in Figure 5.2 and Figure 5.3.

Suppose that t has an innermost sub-term[1] $t_1 = \texttt{Add}_{x \triangleright y}(t_2)$ where t_2 is not of the form $(a,x) \oplus (b,y)$. Let $u \in V^{t_1}$ and $v \in V^{t_1}$ be the unique vertices coloured x and y respectively in G^{t_1} (that is, $\chi^{t_1}(x) = u$ and $\chi^{t_1}(y) = v$). Since t is $\triangleright$-rename-free, u and v were created with colours x and y respectively and they were never recoloured.

In the sub-term t_2, there needs to be a sub-term of the form $t_3 = t_4 \oplus t_5$ such that u is created in t_4 as (a,x) and v is created in t_5 as (b,y). Our idea is to detach v from t_5 and attach it next to u as shown in Figure 5.2 and Figure 5.3. This requires transforming t_4 to t_4' and t_5 to t_5', and finally t_3 to t_3'. We will first describe the transformation, and argue its correctness later.

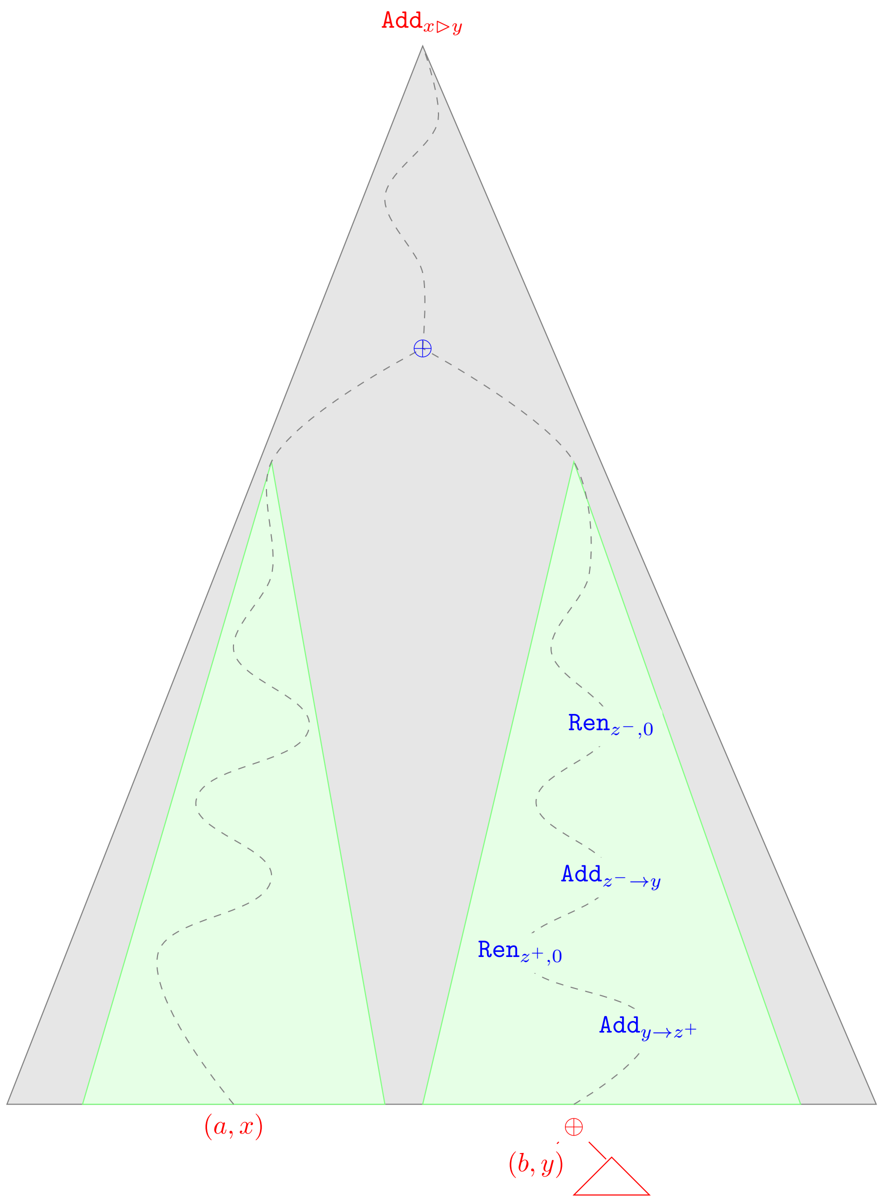
The term t_4' is obtained from t_4 by replacing the leaf (a,x) with $\texttt{Add}_{x \triangleright y}((a,x) \oplus (b,x'))$. All the operations in t_4, including those done on u, are maintained as such in t_4'.

However there are several subtleties on obtaining t_5'. Note that $v \notin V^{t_5'}$. Hence the operations in t_5 involving v need to be postponed until v appears in t_3'

In the sub-term t_5, node v might have got a left neighbour v^- and/or a right neighbour v^+. If there were $\rightarrow$ edge additions between v and its neighbours in t_5, these spurious edge additions are omitted in t_5'. However, we need to keep the vertices v^- or v^+ coloured until the postponed edge additions are done. But perhaps t_5 may involve colouring v^- or v^+ to 0, and their previous colours may be later used by other vertices.

We solve this issue by colouring the vertices v^- and v^+ to two free colours instead of recolouring them to 0. We use y and y' as the free colours.

[1]to be precise, we are referring to one occurrence of a sub-term



$\texttt{Add}_{x \triangleright y}$
$\oplus$
$\texttt{Ren}_{z^-,0}$
$\texttt{Add}_{z^- \to y}$
$\texttt{Ren}_{z^+,0}$
$\texttt{Add}_{y \to z^+}$
(a, x)
$\oplus$
(b, y)

Figure 5.2: A $\triangleright$-rename-free term. Its equivalent $\triangleright$-leafed term is shown on Figure 5.3

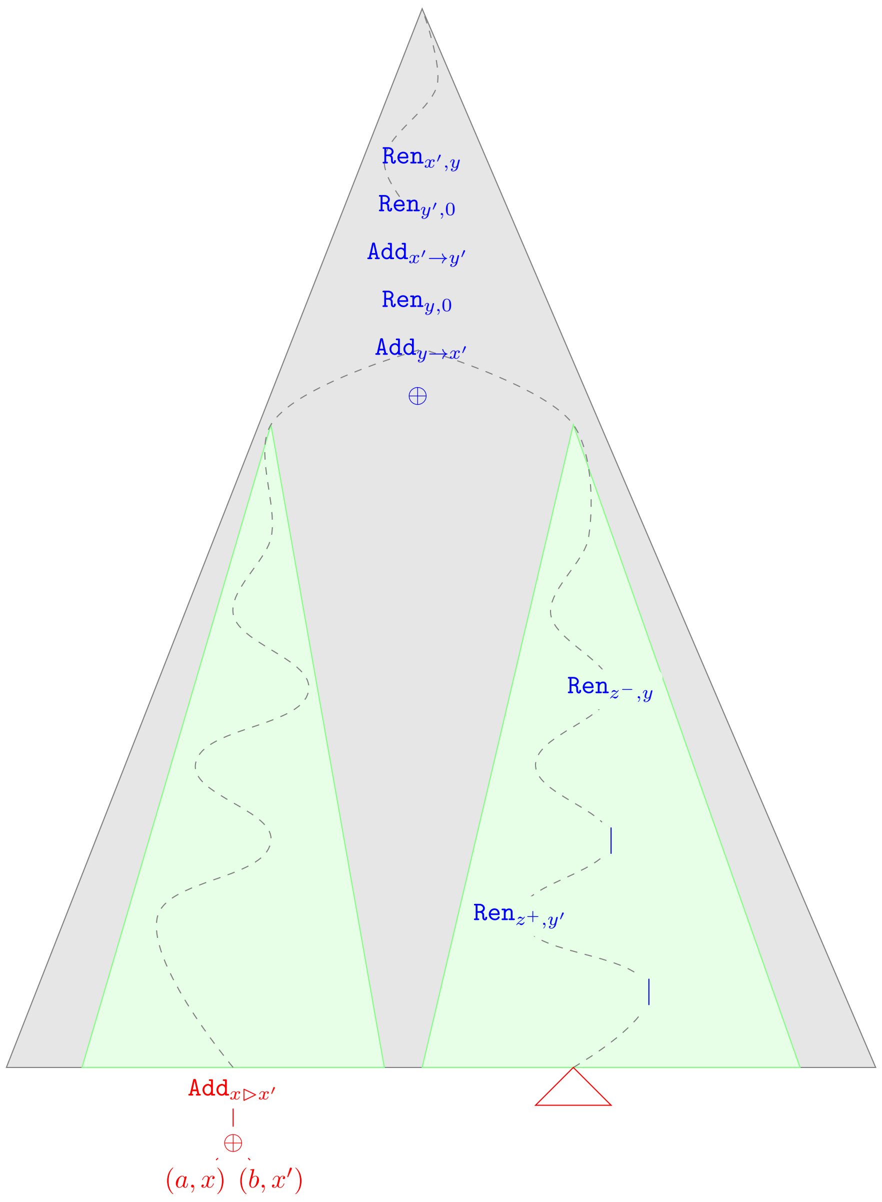


Figure 5.3: A $\rhd$-leafed term corresponding to the term in Figure 5.3. Two additional colours x' and y' are used. Note that these are recoloured to 0, so that these colours are not bound outside the subtree.

Thus t_5' is obtained from t_5 as follows:

1. The leaf (b, y) where vertex v was created is removed. That is, the sub-term $(b, y) \oplus t_6$ is replaced by t_6 in order to get the term t_5^1.

2. If t_5^1 has a sub-term $\mathtt{Add}_{z^- \to y}(t_7)$ such that $\chi^{t_7}(z^-) = v^-$, we replace it by t_7 to get t_5^2. Otherwise, $t_5^2 = t_5^1$.

3. If t_5^2 has a sub-term $\mathtt{Add}_{y \to z^+}(t_8)$ such that $\chi^{t_8}(z^+) = v^+$, we replace it by t_8 to get t_5^3. Otherwise, $t_5^3 = t_5^2$.

4. If t_5^3 has a sub-term $\mathtt{Ren}_{z^-,0}(t_9)$ where $\chi^{t_9}(z^-)$ is v^-, then we replace it by $\mathtt{Ren}_{z,y}(t_9)$ to obtain t_5^4. Otherwise, $t_5^4 = t_5^3$.

5. If t_5^4 has a sub-term $\mathtt{Ren}_{z^+,0}(t_{10})$ where $\chi^{t_{10}}(z^+)$ is v^+, then we replace it by $\mathtt{Ren}_{z,y'}(t_{10})$ to obtain t_5'. Otherwise, $t_5' = t_5^4$.

Note that Figure 5.2 and Figure 5.3 consider all the above cases to be positive. Having obtained t_5' we now describe how to obtain t_3'.

Suppose case 2 and case 3 were positive. Let them be coloured z_1 and z_2 respectively. That is, $\chi^{t_5'}(z_1) = v^-$ and $\chi^{t_5'}(z_2) = v^+$. (Note that, $z_1 = y$ if case 4 above was positive. Otherwise there is no node coloured y in t_5. Similarly $z_2 = y'$ if case 5 above was positive. Otherwise there is no node coloured y' in t_5.) Then

$$t_3' = \mathtt{Ren}_{x',y}(\mathtt{Ren}_{y',0}(\mathtt{Add}_{x' \to z_2}(\mathtt{Ren}_{y,0}(\mathtt{Add}_{z_1 \to x'}(t_4' \oplus t_5')))))$$

The case where only one or none of the neighbours of v are connected to v in G^{t_5} are simpler: One or both of the $\mathtt{Add}_{\to}$ in the expression above are removed depending on the case.

Finally, t_1' is obtained by replacing the sub-term t_3 with t_3'.

Now we argue the correctness of our transformation in the following claim.

Claim 5.19. *1. $G^{t_1} = G^{t_1'}$ and $\chi^{t_1} = \chi^{t_1'}$.*

2. t_1' is a special-tree-term.

3. The term t_1' is $\rhd$-leafed.

4. The term t' in which t_1 is replaced with t_1' in t is strongly-$\rhd$-rename-free.

Proof. 1. By construction of t_1'. All the edge additions have been incorporated, and the nodes have been recoloured properly to match t_1.

2. All we need to argue is that $\chi^{t''}$ is a partial function for any sub-term t'' of t_1'. Since t is strongly-$\rhd$-rename-free, in t_1, the colour x' is not used in the branch of u, and the colour y' is not used in the branch of v. Hence using the colour x' for v in t_4' does not violate injection of χ in t_4'. Similarly y' can colour at most one node in t_5'. Also, y can colour at most one node in t_5', as y coloured at most one node in the branch of v in t_5 and v is not present any more.

3. t_1 has only one $\texttt{Add}_{x \rhd y}$ which was not leafed. Hence, after the transformation all the $\rhd$ edge additions are leafed.

4. Let $\texttt{Add}_{x_1 \rhd y_1}(t'')$ be a non-leaf sub-term of t' adding an edge from vertex u_1 to v_1. Condition SRF1 is trivially true, as the transformation only introduced leafed additions of $\rhd$-edges. Condition SRF2 holds if u_1 (or v_1) is not one of u, v, v^-, v^+. Clearly it cannot be u or v as $\rhd$ is disjoint. If it was v^- or v^+, then this node was not recoloured in t'_1. This is because the recolourings were done only if originally these nodes were recoloured to 0 in t_1 which in turn disallows further edge additions. Thus Condition SRF2 holds.

   If the branches of u_1 and v_1 do not intersect with the branches of u and v in t, then Condition SRF3 holds. This is because the colours x' and y' were recoloured to 0 in t'_3, thus x' and y' are only present on the branches of u and v in t, hence not in the branches of u_1 and v_1. If the branch of u_1 and/or v_1 (call it w) intersects the branch of u and/or v: Since it was never recoloured, $\chi^{t'_1}(w) = z \neq x$ and/or $z \neq y$. Hence z' is not added on the branch of w in the transformation from t_1 to t'_1. Thus Condition SRF3 holds in all cases. □

Thus our transformation moves the $\texttt{Add}_{\rhd}$ operations to the leaf level one by one, from innermost to outermost, maintaining the properties stated in Claim 5.19. □

Now we will put the last piece for proving Theorem 5.15.

Lemma 5.20. *If a CBM has a k-special-tree-term which is* $\rhd$-leafed*, then its split-width is at most $k-1$.*

Proof. Let T be a k-special-tree-term which is $\rhd$-leafed for a CBM $\mathcal{M}$. To every sub-term t of T, we will associate a split-term denoted t-to-$s(t)$ as follows:

- t-to-$s((a,p,x)) = (a,p)$ if (a,p,x) is not a subterm of some $\texttt{Add}_{\cdot \rhd \cdot}$ in t
- t-to-$s(\texttt{Add}_{x \rhd y}((a,p,x) \oplus (b,p',y))) = (a,p) \rhd (b,p')$
- t-to-$s(t_1 \oplus t_2) = t$-to-$s(t_1) \sqcup t$-to-$s(t_2)$
- t-to-$s(\texttt{Add}_{x \to y}(t_1)) = \mathsf{merge}(t$-to-$s(t_1))$
- t-to-$s(\texttt{Ren}_{x,y}(t_1)) = t$-to-$s(t_1)$

Now we claim the following:

Claim 5.21. *1. $\mathcal{M} \in [\![t$-to-$s(T)]\!]$*

2. Width of t-to-$s(T)$ *is at most $k-1$.*

Towards proving the above claim, we first associate a split-CBM to every sub-term t of T, denoted $H^t = (\overline{\mathcal{M}^t}, \xrightarrow{\text{e}}_t)$. We let $\overline{\mathcal{M}^t} = G^t$. Thus it suffices to define the elastic edges $\xrightarrow{\text{e}}_t$. It is defined top-down as follows:

1. $\overset{\mathrm{e}}{\to}_T = \emptyset$
2. if $t = \mathtt{Add}_{x \to y}(t')$ then $\overset{\mathrm{e}}{\to}_{t'} = \overset{\mathrm{e}}{\to}_t \cup \{(\chi^{t'}(x), \chi^{t'}(y))\}$.
3. if $t = \mathtt{Ren}_{x,y}(t')$ then $\overset{\mathrm{e}}{\to}_{t'} = \overset{\mathrm{e}}{\to}_t$.
4. if $t = t_1 \oplus t_2$ then, for $i = 1, 2$,

   $$\overset{\mathrm{e}}{\to}_{t_i} = \{(u,v) \mid u, v \in V^{t_i} \text{ and } (u,v) \in [\![\overset{\mathrm{e}}{\to}_t \cdot (\text{¿}(V^{t_{3-i}})? \cdot \to_t)^*]\!]\}$$

   That is, $\overset{\mathrm{e}}{\to}_{t_i}$ relate a pair of vertices if they are separated by at least one elastic edge in between, and all the vertices in between them must belong to its complementary graph $G^{t_{3-i}}$.

Claim 5.22. $H^t \in [\![t\text{-to-}s(t)]\!]$

Proof. We argue this claim by induction. Notice that the inductive definition of H^t terminates once a sub-term $\mathtt{Add}_{x \rhd y}$ or a leaf is encountered. The split CBMs associated to these terms are uniquely determined. Thus the claim holds for base terms.

For the inductive case

1. If $t = \mathtt{Ren}_{x,y}(t')$, then $H^t = H^{t'}$ and $t\text{-to-}s(t) = t\text{-to-}s(t')$, hence by induction $H^t \in [\![t\text{-to-}s(t)]\!]$.
2. If $t = \mathtt{Add}_{x \to y}(t')$, then $H^t \in \mathsf{merge}(H^{t'}) \subseteq [\![t\text{-to-}s(t)]\!]$
3. If $t = t_1 \oplus t_2$, then $H^t \in H^{t_1} \sqcup\!\sqcup H^{t_2} \subseteq [\![t\text{-to-}s(t)]\!]$

Thus the claim holds by induction. □

Item 1 of Claim 5.21 follows from Claim 5.22 since $H^t = G^t = \mathcal{M}$.

Claim 5.23. *For every sub-term t of T, the vertices participating in $\overset{\mathrm{e}}{\to}_t$ are contained in $\chi^t(\{1, \ldots, k\})$.*

Proof. If $(u,v) \in \overset{\mathrm{e}}{\to}_t$, then there is $t_1 = \mathtt{Add}_{x \to y}(t_2)$ sub-term of T such that t is a sub-term of t_2 and $\chi^{t_2}(x) = u$ and $\chi^{t_2}(y) = v$. Then we know that u (resp. v) is coloured in all G^{t_3} if t_3 is a sub-term of t_2 with u (resp. v) in V^{t_3}. This holds in particular when $t_3 = t$. Thus the claim holds. □

Item 2 of Claim 5.21 follows from the above claim as there can be at most $k-1$ elastic edges between k vertices. Lemma 5.20 follows from Claim 5.21. □

Chapter 6

Discussions

This part of the thesis has introduced the notion of split-width. This is the main technical tool we use to analyse CPDS. It allows us to encode an MSCN as a tree and employ tree-automata techniques for the parametrised verification, yielding decidability. It compares well to other graph parameters such as clique-width and tree-width.

In Chapter 3 we have defined CBMs, split-CBMs and the notion of split-width.

The power of split-width as a tool for under-approximate verification is exploited in Chapter 4. A way to disambiguate a split-term is proposed. This gives an encoding of a split-CBM as a binary tree, with a bijection between the events and the leaves of the tree. The edge relations can be easily recovered. This tree-encoding has opened us the possibility of benefitting from tree-automata techniques for the under-approximate verification of CPDS.

The significance of our results is best exposed in this light of related work. We capture uniformly various restrictions studied for under approximate verification of specific architectures. We also provide a uniform decision procedure for various verification problems. Our generic decision procedure meets the optimal complexity in most cases. Moreover, our technique easily extends to generalisations. The next part (Part III) elaborates these.

In Chapter 5 we have shown that split-width is 'equivalent' to the classical notion of tree-width / clique-width on CBMs. A bound on one implies a linear bound on the other. Thus split-width, while offering a simple inductive way to express a CBM, also is powerful enough to capture any CBM with a bounded tree-width. In Section 6.1 we compare split-width and classical notion of tree-width/split-width in terms of readability and usability.

One could ask numerous interesting questions in this extent. We have not been able to address these questions during this thesis due to lack of time. We will state some[1] important questions in Section 6.2 which we leave open.

[1]It is by no means an exhaustive listing of open questions in this area.

6.1 Split-width versus Tree-width

In Chapter 5 we have seen that split-width is 'equivalent' in power to tree-width / clique-width. This calls for a closer comparison between these metrics.

Tree-width and clique-width are generic graph-theoretic metrics. These are defined for arbitrary graphs, and hence in the case of particular classes of graphs, do not exploit the strict structure of the underlying graph into their definition. In other words, all the edges are treated the same, whether they form a linear / partial order, or preserve some other structure like well-nestedness.

`split-width exploits the underlying graph structure`

Split-width on the other hand implicitly reply on an underlying linear order on each process. This linear order is abstracted away from the definitions, but rather is reflected in the shuffles and merges which preserve an order. Moreover, the $\rhd$ edges appear just at the atomic level. Thus the explicit handling of edges in the case of tree-terms and clique-terms are made implicit in split-terms. Hence one need not worry about edge additions while obtaining a split-term for an MSCN.

Thus, split-width, in a sense, is a notion equivalent to tree-width which is well-tuned for MSCNs. The simplicity in definition can be attributed partly to the underlying structure of the MSCN.

Another feature which makes split-width handier and easier than tree-width is its inductive definition. To prove the split-width of an MSCN, one needs to find a splitting of its $\rightarrow$ edges to get two disconnected components. This can be called a "split" operation. The cost of a split-operation is the number of $\rightarrow$ edges getting split (into an elastic one).

Thus a split operation divides a split-MSCN into two disconnected parts. These two disconnected parts are again split-MSCNs. Note that each of these parts still preserve the order induced by the split by means of an elastic edge.

`divide-and-conquer`

One can then try to recursively split each of these parts independently. Thus it provides a "*divide and conquer*" approach for obtaining a split-term.

A 'split' operation described above corresponds to the inverse of several 'merge' operations. The 'division' is the inverse of the 'shuffle'.

The split-width can also be expressed as a recurrence. It is the maximum amongst the cost of a single split, and the split-width of the sub-parts.

The divide and conquer approach makes it easier to obtain the split-width for an MSCN. This also gives a dynamic programming solution to compute the split-width of a given MSCN. The fact that the sub-parts one has to deal with are also of the same nature (they are also split-MSCNs) makes the analysis easier. One has a better understanding of what sub-objects they are dealing with. In a deeper level, one may also identify these are essentially consecutive chunks of execution trace of some CPDS.

Obtaining a (disambiguated) split-term for an MSCN also suggests a tree-decomposition for it. In fact, a tree-decomposition has the same structure as the split-term. Each node of the split-term corresponds to a bag which contains all the vertices which are the extreme events of some component. Thus a split-term gives a tree-decomposition, which in addition gives some intuitive insights.

The split-term also suggests one way to root the tree-decomposition. The bags contain essentially end-points of blocks of execution-threads. Thus the vertices in a bag may be preceded or succeeded by events which are not yet present in the current subtree.

Notice that, once we move to a tree-decomposition suggested by a split-term, we already lose the information about whether a vertex in the bag has one or none of its linear neighbours. This information is transparent in a split-term. For an arbitrary tree-decomposition (one that is not obtained this way from a split-term), we may only infer that some of the neighbours of a vertex in a bag might be missing (this need not be the case always, as a vertex can be present in a bag just because it is on the path between two other bags which contain this vertex). Moreover, if a neighbour is missing, it could very well be a $\rhd$-neighbour.

On the other hand, a split-term offers an embedding of an MSCN. There is a bijective correspondence between the events of an MSCN and the leaves of the split term. The edges correspond to specific paths and are easily recoverable by a walking tree automaton. This makes reasoning about the MSCN structure on the tree-domain easy, as witnessed by the PDL satisfiability checking procedure.

`embedding`

Thus split-width serves as a way to bound tree-width. It provides a divide-and-conquer approach to obtain a tree-decomposition, and also provides extra insights and meanings to the bags in the tree-decomposition.

The fact that bounded split-width implies bounded tree-width can be argued alternately as:

> MSCNs are bounded degree graphs. The class of bounded split-width MSCNs have a decidable MSO theory (cf. Theorem 4.15). If a class of bounded degree graphs has a decidable MSO theory, then it has bounded clique-width (cf. Fact 5.4). For bounded degree graphs, clique-width is bounded if and only if its tree-width is bounded (cf. Fact 5.11).

Nevertheless, we have described above a constructive way to obtain a tree-decomposition. This allows us to conclude a *linear* bound on tree-width. In fact we have been able to improve the bound on tree-width for multi-pushdown systems under ordering restriction from $\mathfrak{s}2^{\mathfrak{s}}$ to $2^{\mathfrak{s}+1}$ [CGN12a] by demostrating a bound on split-width.

`way to bound tree-width`

However, the converse direction which shows that any class of MSCNs with a bounded tree-width also has a bounded split-width is surprising. If an MSCN has a tree-decomposition which witnesses an arbitrary edge ($\rhd$ or $\rightarrow$) in an arbitrary bag, it indeed has another one in which the $\rhd$ edges are witnessed almost at leaves, and the bags having specific meaning. In fact such a tree-decomposition can be understood as a sequence of divide-and-conquer operamore meaningful tree-decomposition

tions on the MSCN. This more involved direction, gives us new insights about tree-width for MSCNs.

For a closer structural comparison between split-terms and clique-terms, we find special tree-terms as the right intermediate formalisms. Table 6.1 contrasts split-terms and special tree-terms for MSCNs.

Objects	split-terms	special tree-terms
Constant terms	(a, p)	(a, p, c)
Edges	$\rhd$ (restricted to leaves), merge()	$\mathtt{Add}_{c_1 \to c_2}$, $\mathtt{Add}_{c_1 \rhd c_2}$ (unrestricted)
Growth / expansion / union	shuffle ($\sqcup\!\sqcup$) (respecting linear order on the components)	Disjoint union ($\oplus$) (no structure on the components)
Miscellaneous	N.A.	$\mathtt{Ren}_{c_1, c_2}$

Table 6.1: Comparison between split-terms and special tree-terms for MSCNs.

Thus, split-width and tree-width are two equivalent notions. Tree-width has been well-studied and is now classical as opposed to split-width. One might wonder whether we need a new metric when we already have another one which is well-studied, particularly when the new metric is simply equivalent to the old one. We address these concerns with the following analogy.

FO vs. LTL analogy

Consider first-order logic and linear time temporal logics (LTL) over words. These two are equivalent in expressive power. It is rather easy to see that a temporal logic can be expressed in first order logic, as the semantics of every modality is FO definable. The other direction is more involved, but nevertheless, these two formalisms are equal in power.

First order logic is a classical logic which is very well-studied and accepted. Even then, temporal logics adorn a very special place and is very widely used. One of the reasons behind this is the following.

LTL makes reasoning about properties easier and simpler. The first-order logic formulas have the 'order' relation and the free-variables, which one has to keep in mind while writing a formula to state a property. On the other hand, temporal logic formulas make these implicit in their definition. One does not have to worry about free-variables, what it corresponds to, and the underlying linear order.

Notice that, the well established first-order logic is defined for arbitrary graphs and is well studied, whereas LTL is defined only for linear orders. Nevertheless LTL is much more handy once we are studying words (or linear orders).

In the analogy tree-width corresponds to first-order logic which is classical, well-studied and established. The bags in a tree-decomposition (or the colours in a tree-term) are analogous to the free-variables. Just like first-order logic is defined over arbitrary structures, tree-width is defined over arbitrary graphs. On the other hand, split-width is defined only for specific structures of MSCNs, similar to LTL which is defined only for linear orders. The explicit bags of a tree-decomposition are abstracted away in a split-term, just like the free-variables are abstracted away in an LTL formula. The merge and shuffle operations are analogous to the temporal modalities. These operations rely on, and at the same time abstract away, the underlying order in the structure.

Thus to conclude, we believe that split-width is a handier tool for the analysis of MSCNs. However, unlike tree-width/clique-width, split-width is not defined for arbitrary graphs, and cannot be a replacement for the former. Nevertheless, one may benefit from the divide-and-conquer approach and the deeper insights into the underlying structure offered by split-width for the analysis of MSCNs.

6.2 Open Problems

However, there are many interesting questions one could ask in this setting, which we have not considered in this thesis due to shortage of time. These form interesting directions for future research. We list out a few such questions in this section.

Open Question 6.1. *Are the decision procedures for the parametrised problems optimal for CPDS inclusion checking and universality checking?*

One interesting specification formalism which we have not considered is μ-calculus. How do we compute fix-points in tree encodings like a DST. What is the complexity? Can the decision procedure for PDL be extended to capture μ-calculus?

Open Question 6.2. *What is the complexity of the parametrised (by the bound on split-width) model checking a CPDS against a μ-calculus formula?*

What is the complexity of the parametrised satisfiability of μ-calculus?

Another very interesting question is a logical characterisation of recognisability by CPDS. We know that the language of a CPDS can be expressed in the existential fragment of MSO. However, the other direction is more challenging. Given an MSO formula, does there exist a CPDS whose language corresponds to the satisfiable models of the formula? This question cannot be answered affirmatively always, as CPDS as not closed under complementation.

Does it become decidable, if parametrised by a bound on split-width?

Open Question 6.3 (SW-par-MSO-to-CPDS)**.**

Input φ*: an* $\mathsf{MSO}(\mathfrak{A}, \Sigma)$ *formula,*

$k \in \mathbb{N}$.

Question *Does there exist a CPDS* $\mathcal{S}$ *over* $\mathfrak{A}$ *and* Σ *such that*

for every MSCN $\mathcal{M} \in k\text{-}\mathbb{MSCN}(\mathfrak{A}, \Sigma)$,

$\mathcal{M} \in \mathscr{L}(\mathcal{S})$ *if and only if* $\mathcal{M} \models \varphi$*?*

The above question can be asked for various other logical languages. Let L be a logical language describing MSCNs over $\mathfrak{A}$ and Σ.

Open Question 6.4 (SW-par-L-to-CPDS($\mathfrak{A}, \Sigma$))**.**

Input exp*: an expression in the language* L*,*

k*: an integer*

Question *Does there exist a CPDS* $\mathcal{S}$ *such that*

$\mathscr{L}(\mathcal{S}) \cap k\text{-}\mathbb{MSCN}(\mathfrak{A}, \Sigma) = \mathscr{L}(\text{exp}) \cap k\text{-}\mathbb{MSCN}(\mathfrak{A}, \Sigma)$*?*

Open Question 6.5. *Does there exist a non-trivial logical language* L *such that Problem 6.4 becomes decidable?*

We did not address closure under complementation while discussing the closure properties with respect to the bound on split-width. We know, that the language of CPDS are not closed under complementation in general. However, if we impose a bound on split-width, can one effectively complement a CPDS?

Open Question 6.6 (SW-par-CPDS-Complementaion)**.**

Input $\mathcal{S}$*: a CPDS over* $\mathfrak{A}$ *and* Σ*,*

$k \in \mathbb{N}$.

Question *Does there exist a CPDS* $\mathcal{S}'$ *such that*

$\mathscr{L}(\mathcal{S}') \cap k\text{-}\mathbb{MSCN} = k\text{-}\mathbb{MSCN}(\mathfrak{A}, \Sigma) \setminus \mathscr{L}(\mathcal{S})$*?*

Open Question 6.7. *Is Problem 6.6 decidable? What is the precise complexity?*

The above problem would become decidable (with a trivial answer 'yes') if we have a translation from a tree-automaton over valid k-DSTs to a CPDS over the corresponding architecture such that an MSCN is accepted by the CPDS if and only if its k-DST encoding is accepted by the tree-automaton. This gives us another interesting problem at hand.

Open Question 6.8 (SW-par-DST-to-CPDS)**.**

Input *$\mathcal{A}$: a tree-automaton over k-DST encodings of k-$\mathbb{MSCN}(\mathfrak{A},\Sigma)$,*

$k \in \mathbb{N}$.

Question *Does there exist a CPDS $\mathcal{S}$ over $\mathfrak{A}$ and Σ such that*

$\mathscr{L}(\mathcal{S}) = \{[\![T]\!] \mid T \in \mathscr{L}(\mathcal{A})\}$?

Note that the above question asks for $\mathcal{S}$ to be faithful to $\mathcal{A}$ only within k-MSCNs. $\mathcal{S}$ is free to have behaviours whose split-width is more than k. This gives us an even more challenging problem at hand:

Open Question 6.9 (SW-to-CPDS)**.**

Input *$\mathfrak{A}$: an architecture,*

$k \in \mathbb{N}$.

Question *For any set of actions Σ, does there exist*

a CPDS $\mathcal{S}$ over $\mathfrak{A}$ and Σ such that $\mathscr{L}(\mathcal{S}) = k$-$\mathbb{MSCN}(\mathfrak{A},\Sigma)$?

We leave these very interesting questions open. However, to compensate, we consider some distributed controllers for CPDS in Part III. These distributed controllers are implementable, and enforces a bound on split-width. The CPDS, when run synchronously with the distributed controller, is guaranteed to generate only behaviours of bounded split-width.

6.3 Frequently Asked Questions

▶ *Can MSCNs of split-width k be recognised by a CPDS?*

We believe it is not possible to recognise k-$\mathbb{MSCN}(\mathfrak{A},\Sigma)$ by a CPDS. It is in fact a very interesting open problem (cf. Open Question 6.9)

.

▶ *Can $\mathscr{L}(\mathcal{S}) \cap k$-$\mathbb{MSCN}(\mathfrak{A},\Sigma)$ be captured by a syntactic restriction on the CPDS $\mathcal{S}$?*

This is another interesting open problem. We believe it is not possible.

However, we can impose syntactic restrictions on CPDS which will impose the given bound on split-width to the permitted behaviours. We will see examples of such restrictions in the next chapter. However, these may forbid some valid behaviours of the system whose split-width is well within the bound.

▶ *Is there an MSO formula over MSCNs describing MSCNs of split-width k?*

We believe it is not possible to have an MSO description of bounded split-width. However, MSO with fix-points seem more promising in this regard.

▶ *Can split-width be a replacement for tree-width? What are the advantages of tree-width over split-width?*

It depends on what structure one is interested in. Please refer to Section 6.1.

▶ *Are there any complexity gains on using split-width as a verification parameter instead of tree-width?*

Since we have linear translations between the various bounds, we believe there are no theoretical complexity gains. However, we gain a better understanding of the verification procedures. For example, the evaluation of a PDL formula over a split-term is intuitive, whereas it is not so clear what it would mean over a tree-decomposition. Thus, we believe split-width would enable us to find easier decision procedures, even though one may be able to later adapt this procedure, or find new procedure matching our complexity, using tree-width.

▶ *Bounded split-width is necessary for MSO decidability. Can one decide weaker properties, say emptiness, for classes without bounded split-width?*

We do not know. This is another direction for future research.

▶ *What kind of syntactic restrictions on the systems can bounded split-width capture?*

We do not have a precise characterisation. However we will see some very generic classes with bounded split-width in Part III which will give a flavour of the kind of restrictions bounded split-width can capture.

Part III

Distributed Controllers

In Part I we saw systems over generic architectures with concurrents processes and data structures. All verification problems are undecidable for these systems. In Part II we saw a convenient parameter called split-width which allows parametrised verification of these systems.

In this part, we consider the problem of implementing a restriction used for parametrised verification in to the system.

Chapter 7

Need for distributed controllers

Contents

7.1 Need for implementing restrictions

Consider the product development cycle in the hypothetical firm. The firm develops a system which is formally verified with the help of a parameter.

Verification has been performed up to a fixed value of the parameter split-width (the higher, the better). But this means only a subset of the possible behaviours has been verified. This subset is precisely those behaviours admitting an upper bound on their split-width, and can be seen as a 'restriction' on the whole set of behaviours. Thus k-$\mathbb{MSCN}(\mathfrak{A}, \Sigma) \subseteq \mathbb{MSCN}(\mathfrak{A}, \Sigma)$ is a restriction of $\mathbb{MSCN}(\mathfrak{A}, \Sigma)$

Building a complex system with the specifications verified for only a subset of its behaviours (call it a restriction) is dangerous. For safety critical systems, the existence of some unverified runs in the system is perhaps as bad as having the system not verified. Hence the firm would rather like to 'build the restrictions into the system'.

'Building the restriction into the system' is called implementing a restriction. This is the main topic of study of this part of the thesis.

The very generic restriction used by the firm for verification procedures is often a 'theoretical restriction' (like split-width). Split-width poses us some challenges here. *Is the restriction k-$\mathbb{MSCN}$ implementable?* We believe this is in general not possible.

However, we can consider other implementable restrictions. We can ask the parametrised verification problems with respect to any restriction (cf. Section 4.4.3. Let $\mathbb{C}$ be a restriction (or a class of MSCNs). The parametrised verification problems can be stated as follows:

Problem 21 ($\mathbb{C}$-Emptiness)**.**

Input	$\mathcal{S}$: a CPDS over $\mathfrak{A}$ and Σ.
Question	Is $\mathscr{L}_{\mathbb{C}}(\mathcal{S}) = \emptyset$?

Problem 22 ($\mathbb{C}$-Inclusion)**.**

Input	$\mathcal{S}_1$, $\mathcal{S}_2$: two CPDS over $\mathfrak{A}$ and Σ
Question	Is $\mathscr{L}_{\mathbb{C}}(\mathcal{S}_1) \subseteq \mathscr{L}_{\mathbb{C}}(\mathcal{S}_2)$?

Problem 23 ($\mathbb{C}$-Universality)**.**

Input	$\mathcal{S}$: a CPDS over $\mathfrak{A}$ and Σ.
Question	Is $\mathscr{L}_{\mathbb{C}}(\mathcal{S}) = \mathbb{C}$?

Problem 24 ($\mathbb{C}$-MSO-SAT)**.**

Input	φ: an $\mathsf{MSO}(\mathfrak{A}, \Sigma)$ formula.
Question	Does there exist $\mathcal{M} \in \mathbb{C}$ such that $\mathcal{M} \models \varphi$?

Problem 25 ($\mathbb{C}$-PDL-SAT)**.**

Input	ϕ: a $\mathsf{PDL}(\mathfrak{A}, \Sigma)$ sentence.
Question	Does there exist $\mathcal{M} \in \mathbb{C}$ such that $\mathcal{M} \models \phi$?

Problem 26 ($\mathbb{C}$-TL-SAT)**.**

Input	ϕ: a $\mathsf{TL}(\mathfrak{A}, \Sigma)$ sentence.
Question	Does there exist an $\mathcal{M} \in \mathbb{C}$ such that $\mathcal{M} \models \phi$?

Problem 27 ($\mathbb{C}$-MSO-MC)**.**

Input	φ: an $\mathsf{MSO}(\mathfrak{A}, \Sigma)$ formula.
	$\mathcal{S}$: a CPDS over $\mathfrak{A}$
Question	Do all $\mathcal{M} \in \mathscr{L}_{\mathbb{C}}(\mathcal{S})$ satisfy $\mathcal{M} \models \varphi$?

Problem 28 ($\mathbb{C}$-PDL-MC)**.**

Input	ϕ: a $\mathsf{PDL}(\mathfrak{A}, \Sigma)$ sentence.
	$\mathcal{S}$: a CPDS over $\mathfrak{A}$ and Σ
Question	For all $\mathcal{M} \in \mathscr{L}_{\mathbb{C}}(\mathcal{S})$, does $\mathcal{M} \models \phi$?

Problem 29 ($\mathbb{C}$-TL-MC)**.**

Input	ϕ: a $\mathsf{TL}(\mathfrak{A}, \Sigma)$ sentence.
	$\mathcal{S}$: a CPDS over $\mathfrak{A}$ and Σ
Question	For all $\mathcal{M} \in \mathscr{L}_{\mathbb{C}}(\mathcal{S})$, does $\mathcal{M} \models \varphi$?

Theorem 7.1. *Let $\mathbb{C}$ be a class of MSCNs. If $\mathbb{C}$ is MSO definable, and $\mathbb{C}$ has bounded split-width, then the Problems 21 - 29 are decidable.*

Proof. It follows from MSO definability and the bound on split-width, thanks to Theorem 4.15. □

A main challenge now is to come up with good restrictions which are as lenient as possible. This part is oriented towards this direction. We will propose a few restrictions, which generalise and uniformly capture several restrictions studied in the literature for particular architectures. The following chapters will discuss the implementability of these restrictions, and show an upper bound on their split-width.

7.2 Desirable features of a restriction:

The restrictions we consider for verification are generic in the sense that they are not constrained by the architecture or the alphabet or the system. Split-width, for instance, is defined in a uniform way which can be applied to any fixed architecture and alphabet. We call this feature *uniformity*, and a restriction which possesses uniformity is called a uniform restriction. A uniform restriction is one which is defined independent of the architecture, the set of actions and the system.

`uniform`

`independence`

`system independence`

It is desirable that the implementation can also be done uniformly. That is, there is a generic way to impose the restriction, which is independent of the architecture and the system, and the adaptation to the specific case gives the specific implementation.

For example, a generic implementation method could be a "distributed communication protocol". If all the processes in the system follow the protocol while executing their local actions, it may bound the split-width. The protocol could be something like, at most three 'phases' of data-structure access; a phase is constrained to access only one data-structure.[1]

An implementation method which is NOT generic is 'disabling transitions' of the system. This is highly system dependent. One cannot define generic 'rules' for disabling transitions without really analysing the system.

We choose to implement a restriction again as a CPDS. The implementation essentially suggests a family of CPDS, one for each architecture. This CPDS is called a *controller* for the restriction.

The controller will again be defined in a uniform way. The system can be designed independently. Then the *implementation* (or the controlled system) will be the *synchronous product* of the the controller and the system.

We say that a controller realises a restriction if the former is sound and complete with respect to the latter. Realisability is an important concern while choosing a restriction for under-approximate verification. We will now argue the importance of realisability (or, soundness and completeness).

[1]This is only an example protocol. This protocol does not yield decidability for verification.

Let us denote the restriction under which a system has been formally verified by $\mathbb{C}$. $\mathbb{C}$ identifies, or is equivalent to, a subset of MSCNs. For each architecture $\mathfrak{A}$ and set of actions Σ, we have $\mathbb{C} \subseteq \mathbb{MSCN}(\mathfrak{A}, \Sigma)$. A $\mathbb{C}$-restricted CPDS $\mathcal{S}$ admits fewer behaviours. For a CPDS $\mathcal{S}$ over $\mathfrak{A}$ and Σ, the restricted language of $\mathcal{S}$ is denoted $\mathscr{L}_{\mathbb{C}}(\mathcal{S})$ $(= \mathbb{C} \cap \mathscr{L}(\mathcal{S}))$.

Let $\mathcal{C}$ be a controller for the restriction. Let $\mathscr{L}_{\mathcal{C}}(\mathcal{S})$ denote the behaviours of the $\mathcal{C}$-controlled system.

A controller $\mathcal{C}$ *realises* $\mathbb{C}$ if, for all architectures $\mathfrak{A}$ and set of actions Σ and CPDS over $\mathfrak{A}$ and Σ, $\mathscr{L}_{\mathcal{C}}(\mathcal{S}) = \mathscr{L}_{\mathbb{C}}(\mathcal{S})$. Thus the controller must be independent of the system.

`soundness`

The implementation will guarantee safety if $\mathcal{C} \subseteq \mathbb{C}$. Safety properties check whether an unsafe state can be reached in the system. If $\mathbb{C}$ guarantees that an unsafe state cannot be reached, then so does $\mathcal{C} \subseteq \mathbb{C}$.

`completeness`

For liveness properties, we require completeness ($\mathbb{C} \subseteq \mathcal{C}$). Liveness can be thought of as preserving the possibility to execute some action in the future. Suppose $\mathbb{C}$ satisfies some liveness properties. Removing some behaviours from this set may potentially violate this property.

`distributedness`

Implementing a controller as a CPDS makes it *distributed*. This is a very important property. For physically distributed system, a global (or, non-distributed) controller is practically infeasible.

Thus, a controlled system essentially has two fields in its states and data-structure entries: A 'system field' and a 'control field'. The system does not access/modify the 'control fields' and the controller does not access/modify the 'system fields'.

This method of implementation of a restriction has the following advantages.

`MSO-definability`

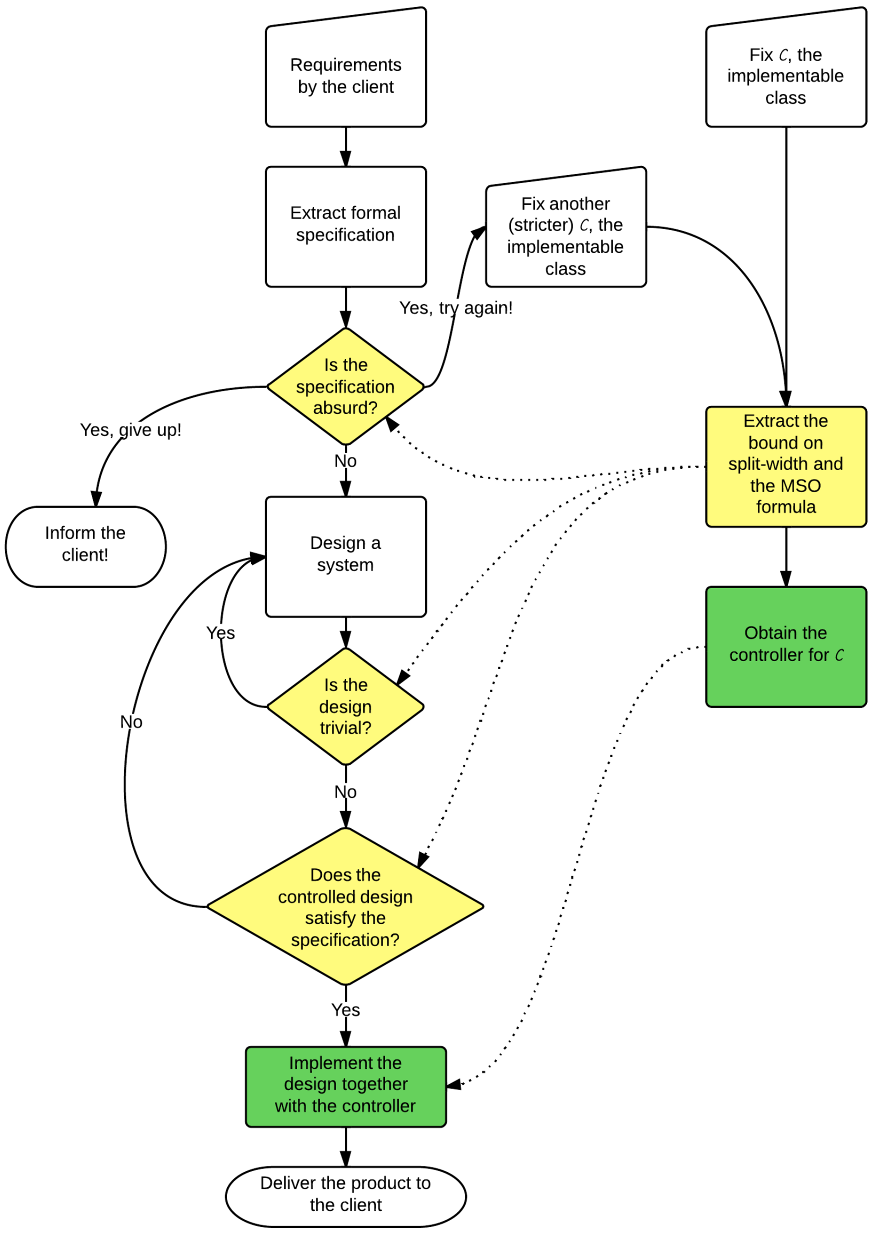
1. MSO definability for free (thanks to Remark 2.19).
2. Better complexity for verification algorithms than working directly with an MSO formula for the restriction.

`privacy/security`

3. Privacy: The controller is not accessing the local states of the processes, or the data-structure entries.
4. Safety: The processes do not modify the states of the controller.
5. Possibility to provide some controllers as "libraries". The controllers available in the library will be uniform. Thus, once a system is designed, it may import the suitable controller from the library and run synchronously with it.

Note that, by asking the controller to be synchronously running with the system, we ensure that no new behaviours are added into the controlled system. Note also that the controlled system is again over the same architecture, the controller does not have private data-structures or independent data-structure accesses. It can only tag the data-structure entries made by the system with necessary control information in the designated 'control field'.

Thus, we have a solution to the problem faced by our hypothetical firm. We could consider various other restrictions which are 1) implementable, 2) MSO definable, and 3) of bounded split-width. If the restriction is MSO definable and possesses bounded split-width, we can employ our decision procedures from Part II for decidability of verification. Of course, the controller realises the restriction in this case. Such a restriction is called a *good* restriction.


Requirements by the client
Fix $\mathcal{C}$, the implementable class
Extract formal specification
Fix another (stricter) $\mathcal{C}$, the implementable class
Yes, try again!
Is the specification absurd?
Yes, give up!
No
Extract the bound on split-width and the MSO formula
Inform the client!
Design a system
Obtain the controller for $\mathcal{C}$
Yes
Is the design trivial?
No
No
Does the controlled design satisfy the specification?
Yes
Implement the design together with the controller
Deliver the product to the client

In this part we will propose some good restrictions which may be added to such a ‘controller library’. Later in this part, we will also show that several well-studied restrictions in the literature for under-approximate verification are subsumed by our library.

Chapter 8

Preliminary notions

Contents

We will propose some generic restrictions which are potential candidates for being good. In the following chapters we will see their definitions, and discuss the implementability and bound on split-width.

First, we introduce some basic notions, which will help us to define, understand and analyse these restrictions. This chapter is dedicated to these necessary preliminaries.

8.1 Contexts: Decompositions and Graphs

A *context* is a consecutive chunk of the execution of a single process. Thus in a context, only one process is active. `context`

A context-decomposition splits an MSCN into different contexts. A *context-decomposition* of an MSCN $\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \delta, \rightarrow, \rhd)$ is a split-MSCN $\mathcal{M}\!\!\!/$ such that $\mathcal{M}\!\!\!/ = (\mathcal{M}, \xrightarrow{\text{e}})$. for some $\xrightarrow{\text{e}}$. Each component of $\mathcal{M}\!\!\!/$ is a *context.* `context-decomposition`

In the following, a split-MSCN is synonymous to a context-decomposition. Likewise, a component is synonymous to a context.

We need to reason about the inter-dependencies of different contexts. This information is abstracted in what is called a context-graph. It is a graph whose vertices are the contexts. The edges relate pairs of contexts related by some data-structure access. We provide a different edge relation for each data-structure.

context-graph

The *context-graph* of a split-MSCN $\mathcal{M}$ denoted $G_{\mathcal{M}}$, is a graph with multiple edge relations. The vertices of this graph are the components of $\mathcal{M}$. There is an edge relation E_d for each data-structure d to link the components related by an access to the data-structure d. An example is given in Figure 8.1. Thus,

$$G_{\mathcal{M}} = (V, (E_d)_{d \in \mathbf{DS}})$$

where

- V is the set of components of $\mathcal{M}$
- $(E_d)_{d \in \mathbf{DS}}$ are edge relations such that for all components $x, y \in V$ and $d \in \mathbf{DS}$, the pair $(x, y) \in E_d$ if and only if there exist events $e_1 \in x$ and $e_2 \in y$ such that $\delta(e_1) = \delta(e_2) = d$ and $(e_1, e_2) \in \rhd$.

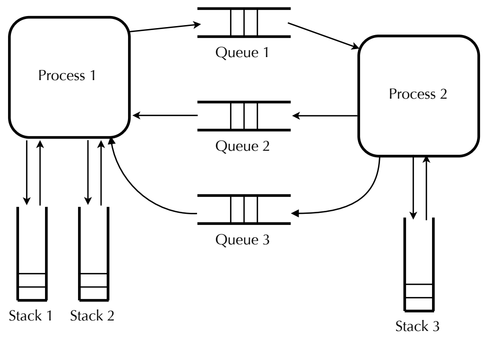
extended-context-graph

An *extended context-graph* is a context-graph which follows a canonical naming of the contexts (or vertices) as follows: The ith context on process p is named (p, i). Thus the canonical naming reveals the pid information and the ordering induced by the elastic edges.

$\preccurlyeq$

We denote by $\preccurlyeq$ the transitive closure of the edge relations of the extended context graph which include the elastic edges. Thus $\preccurlyeq$ is the transitive closure of $\xrightarrow{e} \cup \bigcup_{d \in \mathbf{DS}} E_d$.

Example 8.1. The (extended) context-graph corresponding to a split-MSCN (or context-decomposition) over the following architecture is given in Figure 8.1.



8.2 Classes of context-graphs

$\mathbb{MSCN}_{\mathcal{G}}$

In order to define generic classes of MSCNs we impose restrictions on context-graphs. A class of context-graphs $\mathcal{G}$ defines a class of MSCNs, denoted $\mathbb{MSCN}_{\mathcal{G}}$, as follows: $\mathbb{MSCN}_{\mathcal{G}} = \{\mathcal{M} \mid \mathcal{M}$ has a context decomposition $\mathcal{M}$ such that $G_{\mathcal{M}} \in \mathcal{G}\}$.

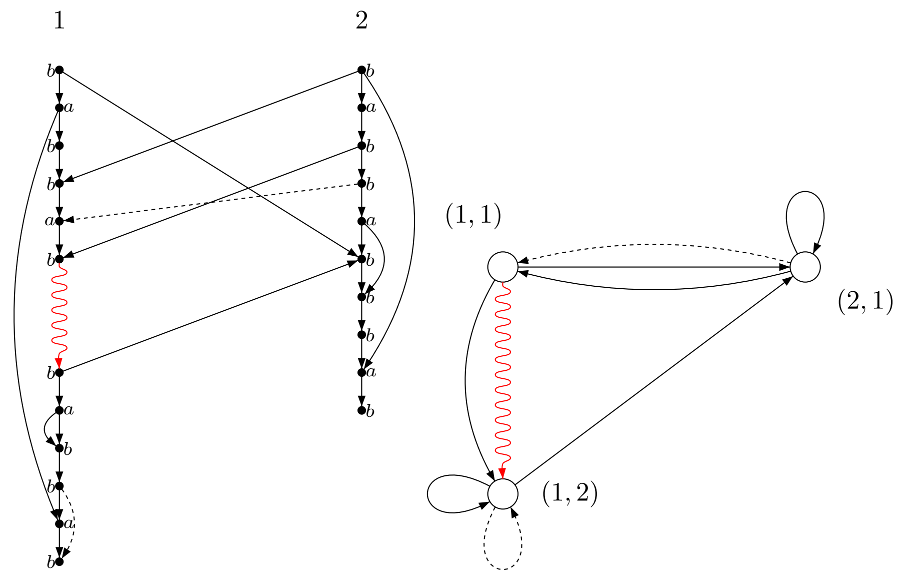


Figure 8.1: A split-MSCN and its (extended) context-graph.

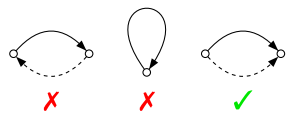


Table 8.1: Directed acyclic yes/no.

8.2.1 Preliminaries

Towards defining classes of context-graphs, we first familiarise with the notion of acyclicity of a context graph.

`directed acyclicity`

A context-graph G is *directed acyclic* if the union of the edge relations does not impose a directed cycle. That is, if $(x, y) \in E_d$ then

- $x \neq y$ (no self loops)
- there are no directed simple path from y to x (using edges from $E = \bigcup_{d \in \mathbf{DS}} E_d$)

Example 8.2. Some (counter-) examples are given in Table 8.1

`undirected acyclicity`

A context-graph G is *undirected acyclic* if the disjoint union of the *undirected* edge relations does not impose an undirected cycle. Note that the disjoint union of the undirected edge relations may yield a graph with several edges between a pair of vertices. This counts as a cycle and is forbidden by the 'undirected acyclicity' condition.

Thus, if $(x, y) \in E_d$ of an undirected acyclic context-graph G then

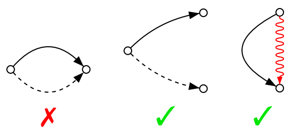


Table 8.2: Undirected acyclic yes/no.

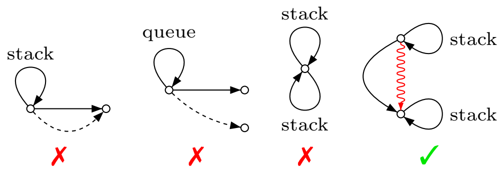


Table 8.3: Undirected acyclic with simple stack self-loops yes/no.

- $x \neq y$ (no self loops)
- there is no *other* undirected simple path from x to y (in $E \cup E^{-1}$)

Thus only one “kind” (with respect to the labelling) of edge is possible between any pair of nodes.

Example 8.3. Some (counter-) examples are given in Table 8.2

`undirected acyclicity with simple stack self-loops`

A context-graph G is *undirected acyclic with simple stack self-loops* if the union of the edge relations does not impose an undirected cycle of length more than one. Moreover the loops must be formed by stack data-structures. That is, if $(x, y) \in E_d$ then

- either $x = y$ and
  - $d \in$ **Stacks** and
  - for all $d' \neq d$, $(x, y) \notin E_{d'}$.
- or $x \neq y$ and
  - there is no other undirected simple path from y to x (in $E \cup E^{-1}$).

Example 8.4. Some (counter-) examples are given in Table 8.3

`directed acyclicity with simple stack self-loops`

A context-graph G is *directed acyclic with simple stack self-loops* if the union of the edge relations does not impose a directed cycle of length more than one. Moreover the loops must be formed by stack data-structures. That is, if $(x, y) \in E_d$ then

- either $x = y$ and
  - $d \in$ **Stacks** and
  - for all $d' \neq d$, $(x, y) \notin E_{d'}$.

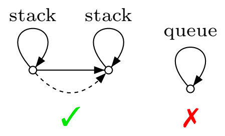


Table 8.4: Directed acyclic with simple stack self-loops yes/no.

- or $x \neq y$ and
  - there is no directed simple path from y to x (in $E = \bigcup_{d \in \mathbf{DS}} E_d$).

Example 8.5. Some (counter-) examples are given in Table 8.4

Now, we are ready to define some classes of context graphs.

8.2.2 Undirected Acyclic Context-Graph (k), $\mathcal{G}_{\mathsf{UA}(k)}$

A context graph G belongs to the class $\mathcal{G}_{\mathsf{UA}(k)}$ if $\mathcal{G}_{\mathsf{UA}(k)}$

DA G is undirected acyclic, and

B G has at most k vertices: $|V| \leq k$.

The next two classes permits simple (restricted) self-loops on otherwise (respectively undirected/directed) acyclic context-graphs. To counterbalance, they forbid read-access to multiple data-structures in the same context.

8.2.3 Undirected Acyclic with Simple Stack Self-loops (k), $\mathcal{G}_{\mathsf{UASL}(k)}$

A context graph G belongs to the class $\mathcal{G}_{\mathsf{UASL}(k)}$ if $\mathcal{G}_{\mathsf{UASL}(k)}$

UASL G is undirected acyclic with simple stack self-loops,

B G has at most k vertices: $|V| \leq k$, and

SR for all contexts x, y, z and data-structures d, d',
if $(y, x) \in E_d$ and $(z, x) \in E_{d'}$, then $d = d'$.

8.2.4 Directed Acyclic with Simple Stack Self-loops (k), $\mathcal{G}_{\mathsf{DASL}(k)}$

A context graph G belongs to the class $\mathcal{G}_{\mathsf{DASL}(k)}$ if $\mathcal{G}_{\mathsf{DASL}(k)}$

DASL G is directed acyclic with simple stack self-loops,

B G has at most k vertices: $|V| \leq k$, and

SR for all contexts x, y, z and data-structures d, d',
if $(y, x) \in E_d$ and $(z, x) \in E_{d'}$, then $d = d'$.

8.2.5 Bounded Scope (k), $\mathcal{G}_{\mathsf{BS}(k)}$

This class is orthogonal to all those described above. There is no bound on the size of the context-graph. But we require that each context accesses at most one data-structure, and, the edges do not extend over unbounded many contexts.

$\mathcal{G}_{\mathsf{BS}(k)}$ A context-graph G belongs to $\mathsf{BS}(k)$ if

SRW Each context accesses at most one data-structure. That is, for all contexts x, y, z and data-structures d, d',
if $(x, y) \in E_d \cup E_d^{-1}$ and $(x, z) \in E_{d'} \cup E_{d'}^{-1}$, then $d = d'$.

DASL G is directed acyclic with simple stack self-loops,

The condition DASL implies that G enriched with the elastic edges between consecutive components is a partial order. That is, $\preccurlyeq$ (page. 122) is a partial order. The strict partial order is denoted $\prec$. If $x \prec y$, we say that x is in the past of y. Note that, the partial order $\prec$ also takes the elastic edges into account.

We further require the following condition.

BS The number of contexts between the source and target of any edge in G is bounded by k. That is, if $(x, y) \in E_d$ for some data-structure $d \in \mathbf{DS}$, then the number of contexts z such that, z is in the past of y but not in the past of x, is bounded by k : $|\{z \mid (z \prec y) \wedge \neg(z \prec x)\}| < k$.

Remark 8.6. We could replace the condition DASL with a simpler condition SL:

SL G has only stack self-loops. That is, for all contexts x, if $(x, x) \in E_d$ then d is a stack.

Condition DASL implies SL. Conditions SL and SRW implies that the loops are indeed simple stack self-loops. The restrictive nature of the contexts (SRW + SL) together with the directed acyclicity of MSCNs imply that any context graph satisfying SRW + SL is always directed acyclic. Hence condition SRW + SL is equivalent to SRW + DASL.

Classes of MSCNs We are now ready to define and study some classes of MSCNs based on the classes of context-graphs defined above. As we had mentioned before, a class $\mathcal{G}$ of context-graphs defines a class of MSCNs denoted $\mathcal{G}$-$\mathbb{MSCN}$ $\mathcal{G}$-$\mathbb{MSCN}$. We will examine some such classes more closely in the following chapters.

Chapter 9

Undirected Acyclic Context-Graph (k), $\mathsf{UA}(k)$

Contents

9.1 Definition and Examples

An MSCN $\mathcal{M}$ belongs to the class $\mathsf{UA}(k)$ if it has a context-decomposition $\mathcal{M}$ such that $G_{\mathcal{M}} \in \mathcal{G}_{\mathsf{UA}(k)}$. That is, $\mathsf{UA}(k) = \mathcal{G}_{\mathsf{UA}(k)}$-$\mathbb{MSCN}$. $\mathsf{UA}(k)$

The class $\mathsf{UA}(k)$ does not impose any restriction on contexts. In a context, a process may write to and read from all the data-structures, as long as it avoids cycles.

Example 9.1. All the MSCNs generated by communicating finite-state machines over undirected acyclic architectures belong to this class. If there are n processes in the architecture, then all behaviours of systems over such an architecture belong to $\mathsf{UA}(n)$. The trivial context-decomposition (with no elastic edges) is a witness.

Example 9.2. The class $\mathsf{UA}(k)$ also permits non-trivial behaviours from *cyclic* architectures. An example is given in Figure 9.1, where both the MSC and a context-decomposition witnessing the membership in $\mathsf{UA}(k)$ are depicted.

As witnessed by the example, the class $\mathsf{UA}(k)$ also permits behaviours that are *not existentially bounded*. That is, they need not admit a linearisation in which the data-structure size is bounded.

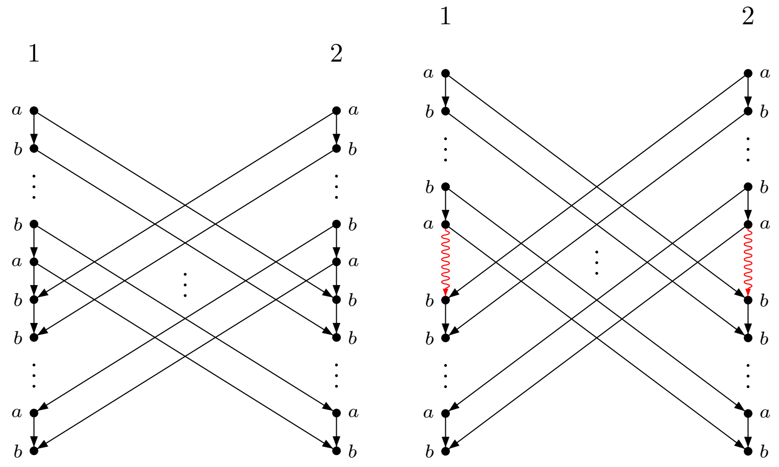


Figure 9.1: An MSC language which is not existentially bounded, and a context-decomposition witnessing membership in $\mathsf{UA}(4)$. Note that the underlying architecture is cyclic.

Example 9.3. Consider a process with a self-queue which is simulating a Turing machine. It scans a configuration left-to-right while enqueuing it. It reads a configuration from the queue, and simultaneously enqueue the next configuration to the queue. Since the queue is unbounded, it may simulate Turing machines with arbitrary long configurations. It can also simulate an arbitrary number of computation steps. This hints that such a system renders all verification problems undecidable.

However, if we limit the computation to only k steps, we gain decidability for the verification problems. Such computations fall in the class of $\mathsf{UA}(k)$. Figure 9.2 depicts the encoding of a 5-step computation of an unbounded Turing machine, and a context-decomposition witnessing its membership in $\mathsf{UA}(5)$.

Remark 9.4. We can convince ourselves that both the conditions UA and B are necessary for decidability.

As suggested by Example 9.3 if $G_{\mathcal{M}}$ is not bounded, it is again Turing powerful.

If we relax undirected acyclicity with weaker directed acyclicity on the context-graph, we also lose decidability. To see this, consider an architecture with two processes and two queues from the first process to the second. We can easily encode solutions to Post's Correspondence Problem as MSCNs over this architecture. Given an instance of a PCP $(u_1, v_1), \ldots (u_k, v_k)$, the first process non-deterministically chooses a sequence of indices i and enqueues u_i in the first queue, and v_i in the second queue. The second process dequeues alternately from the two queues verifying that they span the same word. Notice that the MSCN encoding a solution to the PCP admits a directed acyclic context graph with just two contexts.

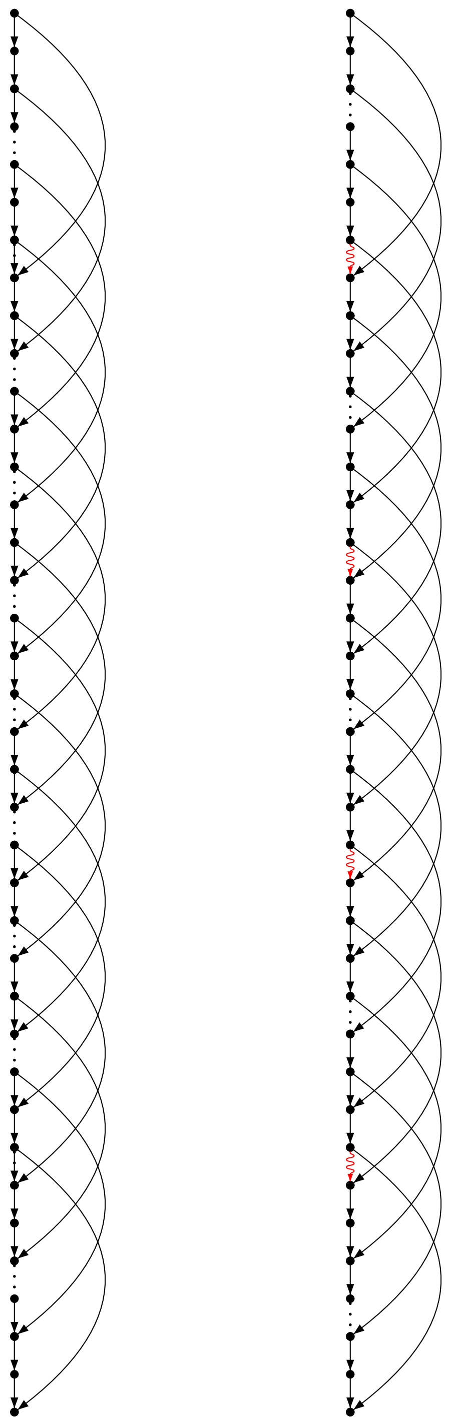


Figure 9.2: An MSCN encoding k computation steps of an unbounded Turing machine, and a context-decomposition witnessing its membership in $\mathsf{UA}(5)$.

If we keep undirected acyclicty, but allow simple stack loops, we are again compromising decidability. Two processes each with one stack, communicating in one direction via a queue between them can check the intersection of two context-free languages. The behaviours of such systems are MSCNs which would satisfy undirected acyclicity condition with two contexts and have only simple stack loops.

9.2 MSO Definability and Bounding Split-width

The class $\mathsf{UA}(k)$ is EMSO definable and admits bounded-split-width.

Lemma 9.5. *The class* $\mathsf{UA}(k)$ *is* EMSO *definable.*

Proof. The EMSO formula has k existentially quantified second-order variables to represent the contexts. The formula asserts that every event belongs to exactly one context, and that the contexts are contiguous. Then it asserts that there are no self loops, and that there are no undirected cycles in the context-graph. The latter can be said in the first-order fragment as we need to rule out only cycles of length at most k. □

Theorem 9.6. *If an MSCN* $\mathcal{M}$ *is in* $\mathsf{UA}(k)$*, then the split-width of* $\mathcal{M}$ *is at most* $k+1$*. Moreover, it is word-like.*

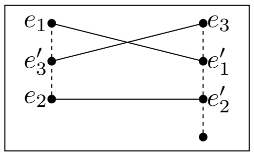
We first prove a couple of preliminary lemmas before proving the theorem.

brink

A *brink* is an extreme event of any component of a split-MSCN. That is, it is either the first event, or the last event of any component.

Lemma 9.7. *In any split-MSCN* $\mathcal{M}$*, there do not exist events* $e_1, e_2, e_3, e'_1, e'_2, e'_3$ *all different and a data-structure* d *such that all the following hold.*

1. $\delta(e_i) = d = \delta(e'_i)$ *for each* $i \in \{1,2,3\}$
2. *for each* $i \in \{1,2,3\}$ *either* $e_i \rhd e'_i$ *or* $e'_i \rhd e_i$
3. e_1*,* e_2 *and* e'_3 *belong to the same component*
4. e'_1*,* e'_2 *and* e_3 *belong to the same component*
5. e_i *is a brink for each* $i \in \{1,2,3\}$



Proof. Such patterns are forbidden in a split-MSCN as otherwise it will violate the data-structure access policy (like LIFO or FIFO). Without loss of generality we may assume that $e_1 \rightarrow^* e'_3 \rightarrow^* e_2$.

First of all, we observe that e_1, e'_3 and e_2 are either all write events, or all read events. If both the components mentioned in the Lemma belong to the same process, since the three events are related to the other component by $\rhd$ edge, they all must be write events if they are in the first component, and they all must be read events if they belong to the second component. If the components mentioned in the Lemma belong to different processes, then the process of e_1 (which is same as $\mathsf{pid}(e'_3)$ and also $\mathsf{pid}(e_2)$) is either $\mathsf{Writer}(d)$ or $\mathsf{Reader}(d)$. In the former case, all three events are writes, and in the latter case they are all read events.

Suppose $d \in \mathbf{Queues}$. Since $e_1 \to^* e'_3 \to^* e_2$, we deduce that $e'_1 \to^* e_3 \to^* e'_2$ from the FIFO policy on queues. Suppose $d \in \mathbf{Stacks}$. According to the LIFO policy on stacks, we deduce that $e'_2 \to^* e_3 \to^* e'_1$. Both these cases contradict e_3 being a brink. □

Lemma 9.8. *If $\mathcal{M}$ is a split-MSCN whose context-graph $G_{\mathcal{M}}$ is finite and undirected acyclic, then*

- *either $\mathcal{M}$ has a brink not taking part in the $\rhd$ relation,*
- *or $\mathcal{M}$ has two brinks connected by a $\rhd$ edge.*

Proof. Suppose all the brinks of $\mathcal{M}$ take part in $\rhd$. In other words, $\mathcal{M}$ does not have a brink not taking part in $\rhd$ relation. We will argue that $\mathcal{M}$ has two brinks connected by a $\rhd$ edge.

In order to witness two brinks connected by a $\rhd$ edge, we start from a random component x_1, and descend to a component x_2 connected to one of its brinks (a brink-neighbour). If there is a $\rhd$ edge between the two brinks of x_2, or if there is a $\rhd$ edge between a brink of x_1 and a brink of x_2, we obtain a witnessing $\rhd$ edge.

Otherwise we descend to a brink-neighbour x_3 which is different from x_1. Such a brink neighbour x_3 exists, thanks to Lemma 9.7. The component x_3 has not been visited before, thanks to undirected acyclicity. This procedure eventually terminates by witnessing a $\rhd$ edge between brinks, as we have only finitely many components. □

Remark 9.9. Lemma 9.8 does not hold if we replace the assumption of undirected acyclicity by weaker directed acyclicity. A counter-example is shown in Figure 9.3. Note that the communication architecture is a directed acyclic graph with at most one queue between an ancestor and a descendant, and there are neither stacks nor self-loops. Moreover, at most one data-structure is read in any context.

The proof of Theorem 9.6 is based on the following lemma.

Lemma 9.10. *Let $\mathcal{M}$ be a split-MSCN such that $G_{\mathcal{M}} \in \mathcal{G}_{\mathsf{UA}(k)}$ (that is, undirected acyclic and has at most k vertices). Then there is a split-term $s_{\mathcal{M}}$ of width at most $k+1$ such that $\mathcal{M} \in [\![s_{\mathcal{M}}]\!]$. Moreover, $s_{\mathcal{M}}$ is word-like.*

Proof. The base cases are when $\mathcal{M}$ is a single event, or when $\mathcal{M}$ is $e \rhd e'$. The split-terms corresponding to the base cases are atomic split-terms.

If the split-MSCN $\mathcal{M}$ has a brink e which is not taking part in $\rhd$ relation, then $\mathcal{M} \in \mathsf{merge}(s_{\mathcal{M}'} \sqcup s_e)$ where $\mathcal{M}'$ is $\mathcal{M}$ without e. $\mathcal{M}'$ is again a split-MSCN such that $G_{\mathcal{M}'} \in \mathcal{G}_{\mathsf{UA}(k)}$. The split-MSCNs in $[\![s_{\mathcal{M}'} \sqcup s_e]\!]$ have at most $k+1$ components. Thus the number of elastic edges is at most k which does

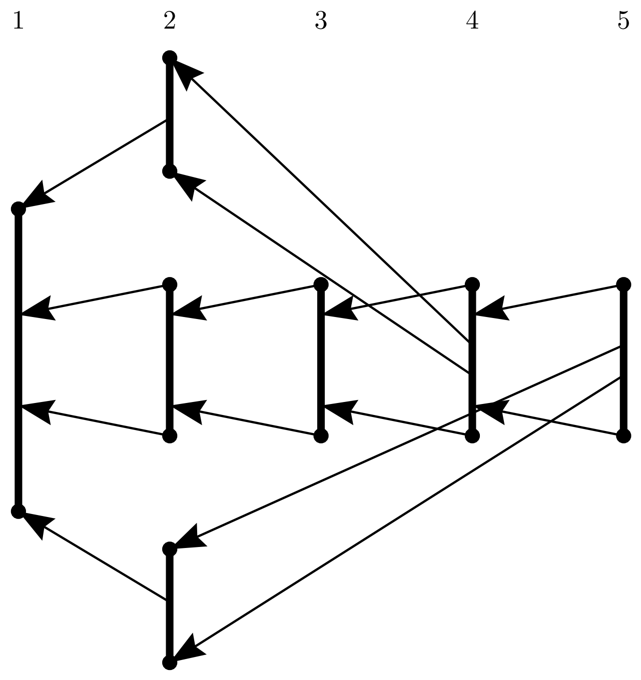


Figure 9.3: Directed acyclicity is not enough to remove $\rhd$-edges one at a time!

not exceed the bound of $k+1$. By induction (notice that $\mathcal{M}'$ has fewer events than $\mathcal{M}$), $\mathcal{M}'$ has split-width at most $k+1$, and hence $\mathcal{M}$ has split-width at most $k+1$.

If the split-MSCN $\mathcal{M}$ has two brinks e and e' such that $(e, e') \in \rhd$, then $\mathcal{M} \in \mathsf{merge}(\mathsf{merge}(s_{\mathcal{M}'} \sqcup s_{(e \rhd e')}))$ where $\mathcal{M}'$ is $\mathcal{M}$ without $e \rhd e'$. Again, $G_{\mathcal{M}'} \in \mathcal{G}_{\mathsf{UA}(k)}$. The split-MSCNs in $[\![(s_{\mathcal{M}'} \sqcup s_{(e \rhd e')})]\!]$ have at most $k+2$ components, and hence at most $k+1$ elastic edges. By induction, $\mathcal{M}'$ has split-width at most $k+1$, and hence $\mathcal{M}$ has split-width at most $k+1$.

In fact, the inductive proof of Lemma 9.10 is complete, thanks to Lemma 9.8.

Notice also that the shuffle expressions have one argument of the form either $s_{(e \rhd e')}$ or s_e. Thus the split-term $s_{\mathcal{M}}$ is word-like. $\square$

Now we are ready to prove Theorem 9.6.

Proof of Theorem 9.6. Let $\mathcal{M}$ be a context-decomposition of $\mathcal{M}$ witnessing the membership in $\mathsf{UA}(k)$. Since $\mathcal{M}$ satisfies $G_{\mathcal{M}} \in \mathcal{G}_{\mathsf{UA}(k)}$, from Lemma 9.10 we have a split-term $s_{\mathcal{M}}$ of width at most $k+1$ with $\mathcal{M} \in [\![s_{\mathcal{M}}]\!]$. We have $\mathcal{M} \in [\![s_{\mathcal{M}}]\!]$ where $s_{\mathcal{M}} = \mathsf{merge}(\ldots \mathsf{merge}(s_{\mathcal{M}}) \ldots)$ and this proves the bound claimed in Theorem 9.6. Since $s_{\mathcal{M}}$ is word-like, the split-term $s_{\mathcal{M}}$ is also word-like. $\square$

9.3 Controller

The class $\mathsf{UA}(k)$ admits a CPDS as a controller.

In fact any class based on a 'finite context-graph' characterisation for some distinguishable contexts admits a finite state CPDS as a controller. A class of MSCNs is defined as those admitting a context-decomposition with the context-graph belonging to a set $\mathcal{G}$. The set $\mathcal{G}$ is a subset of context-graphs with at most k vertices. The classes of context graphs $\mathcal{G}_{\mathsf{UA}(k)}$, $\mathcal{G}_{\mathsf{UASL}(k)}$ and $\mathcal{G}_{\mathsf{DASL}(k)}$ are examples of such $\mathcal{G}$.

We will now describe the controller for a class of MSCNs based on a class of context-graphs $\mathcal{G}$ (that is, $\mathcal{G}$-$\mathbb{MSCN}$). As particular cases, this will give us the controller for the class $\mathsf{UA}(k)$, and also the classes $\mathsf{UASL}(k)$ and $\mathsf{DASL}(k)$ which we will study in the forthcoming chapters.

The controller for $\mathcal{G}$-$\mathbb{MSCN}$ is based on the following policy: Guess a suitable context-graph from $\mathcal{G}$ in the beginning, and along the run, guess a context-decomposition on-the-fly to match the initially guessed context-graph. The global accepting state makes sure that each local process has guessed the same context-graph.

The local state of such a controller has

1. the context-graph guessed in the beginning,
2. information about which context the current event belongs to (as guessed on-the-fly along the run), and
3. information about which all nodes and edges in the context-graph have been witnessed by the run so far.

The local state of the controller for process p is a tuple $\langle G, i, F\rangle$ where

1. $G = (V, (E_d)_{d\in\mathbf{DS}})$ is an extended context-graph from the set $\mathcal{G}$. Recall that, an extended context-graph follows the canonical naming of the contexts which manifests the elastic edges.

2. i is an integer from $\{1, \ldots, k\}$ such that $(p, i) \in V$.

3. $F = (F_d)_{d\in\mathbf{DS}}$ where $F_d \subseteq E_d$, and if $(v_1, v_2) \in F_d$, then $v_2 \preccurlyeq (p, i)$ in G. Recall that $\preccurlyeq$ is the transitive closure of the edge relations of the extended context graphs which include the elastic edges.

G is the guessed extended context-graph. (p, i) indicates the current context. Intuitively, the run so far has witnessed i contexts on process p. Every process tallies the edges of E_d, if it reads from data-structure d. The third component of the local state serves this purpose. It stores the edges it has witnessed so far. Thus every edge set E_d is verified by $\mathsf{Reader}(d)$.

The data-structure entries are tagged with the local state of the write event. In addition the controller has a special control location ℓ_{in} used for the initial state. Recall that a CPDS has a single global initial state.

The finite set of control locations Locs of the controller contains the set of local states, the set of data-structure tags and ℓ_{in}.

The transitions Trans of the controller is a tuple $(\text{Trans}_p)_{p\in\mathbf{Procs}}$, where Trans_p is the set of local transitions of the local controller for process p.

The transitions of the following form are valid initial transitions.

$$(\ell_{\text{in}}, a, \langle G, 1, \emptyset\rangle) \in \text{Trans}_{p:int} \tag{9.1}$$

$$(\ell_{\text{in}}, a, \langle G, 1, \emptyset\rangle, \langle G, 1, \emptyset\rangle) \in \text{Trans}_{p\to d} \tag{9.2}$$

$$(\ell_{\text{in}}, \langle G, i', F'\rangle, a, \langle G, 1, F\rangle) \in \text{Trans}_{p\leftarrow d} \tag{9.3}$$

$$\text{where } F_{d'} = \begin{cases} F'_{d'} & \text{if } d \neq d' \\ \{(\mathsf{Writer}(d), i'), (p, 1)\} & \text{if } d = d' \end{cases}$$

The extended context-graph is guessed in the initial transition. The first event must belong to the first context. Moreover, if the first event is a read event, then the set F must be updated. Further, we have the following transitions.

$$(\langle G, i, F\rangle, a, \langle G, i, F\rangle) \in \text{Trans}_{p:int} \tag{9.4}$$

$$(\langle G, i, F\rangle, a, \langle G, i+1, F\rangle) \in \text{Trans}_{p:int} \tag{9.5}$$

$$(\langle G, i, F\rangle, a, \langle G, i, F\rangle, \langle G, i, F\rangle) \in \text{Trans}_{p\to d} \tag{9.6}$$

$$(\langle G, i, F\rangle, a, \langle G, i+1, F\rangle, \langle G, i+1, F\rangle) \in \text{Trans}_{p\to d} \tag{9.7}$$

$$(\langle G, i, F\rangle, \langle G, i', F'\rangle, a, \langle G, i, F''\rangle) \in \text{Trans}_{p\leftarrow d} \tag{9.8}$$

$$\text{where } F''_{d'} = \begin{cases} F_{d'} \cup F'_{d'} & \text{if } d \neq d' \\ F_{d'} \cup F'_{d'} \cup \{(\mathsf{Writer}(d), i'), (p, i)\} & \text{if } d = d' \end{cases}$$

$$(\langle G, i, F\rangle, \langle G, i', F'\rangle, a, \langle G, i+1, F''\rangle) \in \text{Trans}_{p\leftarrow d} \tag{9.9}$$

$$\text{where } F''_{d'} = \begin{cases} F_{d'} \cup F'_{d'} & \text{if } d \neq d' \\ F_{d'} \cup F'_{d'} \cup \{(\mathsf{Writer}(d), i'), (p, i+1)\} & \text{if } d = d' \end{cases}$$

The transitions (9.1), (9.4) and (9.5) do not access any data-structure. The transitions (9.2), (9.6) and (9.7) write the context-number (given in the target state of the transition) into the data-structure. The transitions (9.3), (9.8) and (9.9) read the context-number (i') of the corresponding send event from the data-structure, and update the set of witnessed edges.

The transitions (9.4), (9.6) and (9.8) provide the choice to keep the same context for the new event, while the other transitions let the new event to mark the start of a new context.

The values of i and i' are bounded as per G. In any case we have $i' < k$. Moreover $i < k$ in (9.5), (9.7) and (9.9) and $i \leq k$ in (9.4), (9.6) and (9.8).

Notice that the extended context-graph is guessed in the initial transition. This guess cannot be changed along the run. The transitions (9.4) – (9.9) preserve the graph G.

Also notice that the transitions dictated by (9.1) – (9.9) only exist if all states are consistent. In particular, the edges added in (9.3), (9.8) and (9.9) must be in E_d.

A tuple $(\langle G^1, i^1, F^1\rangle, \ldots \langle G^{\mathfrak{p}}, i^{\mathfrak{p}}, F^{\mathfrak{p}}\rangle)$ is a global accepting state if

1. $G^1 = \cdots = G^{\mathfrak{p}}$,
2. for each $p \in \mathbf{Procs}$, (p, i^p) is the last context on process p in G^p, and
3. for each $d \in \mathbf{DS}$, $E_d = F^p_d$ where $p = \mathsf{Reader}(d)$.

Notice that the global accepting state guarantees that all the local controllers have guessed the same graph G. Moreover, it makes sure that the guessed graph is witnessed entirely. This defines the set $\mathsf{Locs}_{\mathrm{fin}}$.

The global initial location is $\mathsf{Locs}_{\mathrm{in}} = (\underbrace{\ell_{\mathrm{in}}, \ldots, \ell_{\mathrm{in}}}_{\mathfrak{p}})$.

The above controller for the class $\mathcal{G}$-$\mathbb{MSCN}$ is denoted $\mathcal{S}_{\mathcal{G}\text{-}\mathbb{MSCN}}$. It is a CPDS $(\mathsf{Locs}, \mathrm{Trans}, \mathsf{Locs}_{\mathrm{in}}, \mathsf{Locs}_{\mathrm{fin}})$ where the elements are as described above. $\mathcal{S}_{\mathcal{G}\text{-}\mathbb{MSCN}}$

The number of states of this controller is exponential in k, $\mathfrak{d}$ and $\mathfrak{p}$. More precisely, $|\mathsf{Locs}| = 2^{\mathcal{O}(\mathfrak{d} \times \mathfrak{p} \times k^2 \log k)}$.

Remark 9.11. The above generic controller gives controllers for the special cases of $\mathsf{UA}(k)$, $\mathsf{UASL}(k)$ (Chapter 10) and $\mathsf{DASL}(k)$ (Chapter 11). To obtain the controller for $\mathsf{UA}(k)$ (resp, $\mathsf{UASL}(k)$, $\mathsf{DASL}(k)$) we may replace $\mathcal{G}$ with $\mathcal{G}_{\mathsf{UA}(k)}$ (resp. $\mathcal{G}_{\mathsf{UASL}(k)}$, $\mathcal{G}_{\mathsf{DASL}(k)}$) and $\mathcal{G}$-$\mathbb{MSCN}$ with $\mathsf{UA}(k)$ (resp. $\mathsf{UASL}(k)$, $\mathsf{DASL}(k)$).

We show below that the controller $\mathcal{S}_{\mathcal{G}\text{-}\mathbb{MSCN}}$ is sound and complete for the class $\mathcal{G}$-$\mathbb{MSCN}$. That is, if an MSCN $\mathcal{M}$ is accepted by the controller, then it belongs to $\mathcal{G}$-$\mathbb{MSCN}$. Conversely, if $\mathcal{M} \in \mathcal{G}$-$\mathbb{MSCN}$, then it is accepted by the controller $\mathcal{S}_{\mathcal{G}\text{-}\mathbb{MSCN}}$. In other words, $\mathscr{L}(\mathcal{S}_{\mathcal{G}\text{-}\mathbb{MSCN}}) = \mathcal{G}$-$\mathbb{MSCN}$.

In particular, $\mathscr{L}(\mathcal{S}_{\mathsf{UA}(k)}) = \mathsf{UA}(k)$, $\mathscr{L}(\mathcal{S}_{\mathsf{UASL}(k)}) = \mathsf{UASL}(k)$, and $\mathscr{L}(\mathcal{S}_{\mathsf{DASL}(k)}) = \mathsf{DASL}(k)$.

Lemma 9.12. $\mathscr{L}(\mathcal{S}_{\mathcal{G}\text{-}\mathbb{MSCN}}) \subseteq \mathcal{G}$-$\mathbb{MSCN}$

Proof. Suppose $\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \delta, \rightarrow, \rhd) \in \mathscr{L}(\mathcal{S}_{\mathcal{G}\text{-}\mathbb{MSCN}})$. Then the controller $\mathcal{S}_{\mathcal{G}\text{-}\mathbb{MSCN}}$ has an accepting run $(\text{c-loc}, \text{d-loc})$ on $\mathcal{M}$. The mapping c-loc uniquely determines a context-decomposition $\mathcal{A}$ for $\mathcal{M}$ as follows. All the vertices on process p labelled $\langle G, i, F\rangle$ by c-loc for some G and F constitutes the context (p, i). The graph appearing in the c-loc mapping is precisely the extended context-graph $G_{\mathcal{A}}$ of $\mathcal{A}$: The run registers in its state every context and edge it witnessed so far, and the global acceptance condition guarantees this. Since the graphs permitted in the control locations, and in particular $G_{\mathcal{A}}$, are from $\mathcal{G}$ (by definition), $\mathcal{A}$ serves as a witnessing context-decomposition for the membership of $\mathcal{M}$ in $\mathcal{G}$-$\mathbb{MSCN}$. □

Lemma 9.13. $\mathcal{G}$-$\mathbb{MSCN} \subseteq \mathscr{L}(\mathcal{S}_{\mathcal{G}\text{-}\mathbb{MSCN}})$

Proof. Let $\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \delta, \rightarrow, \rhd) \in \mathcal{G}$-$\mathbb{MSCN}$, and let $\mathcal{A}$ be a witnessing context-decomposition. We can extract an accepting run $(\text{c-loc}, \text{d-loc})$ of $\mathcal{S}$ on $\mathcal{M}$ as follows. For every event $e \in \mathcal{E}$, c-loc maps it to the location $(G_{\mathcal{A}}, i, F)$

where i is the context-number of event e as per $\cancel{\mathcal{M}}$ (cf. the canonical naming of the contexts in the extended context-graph $G_{\cancel{\mathcal{M}}}$), and F contains the $\rhd$ edges which are in the past of e: the triplets as uniquely determined by $\cancel{\mathcal{M}}$ and e: $(x_1, x_2) \in F_d$ if and only if there exist $e_1, e_2 \in \mathcal{E}$ such that 1) $e_1 \rhd e_2$, 2) $e_2 \leq e$ (that is, $(e_2, e) \in (\to \cup \rhd)^*$), 3) $\delta(e_2) = d$ $(= \delta(e_1))$ and 4) for $b \in \{1, 2\}$ the event e_b belongs to context x_b. The d-loc map of a write event e is same as its c-loc map (and it is $\perp$ for the other events). This defines a valid run of $\mathcal{S}$. We can verify that the (c-loc, d-loc) mapping of any local neighbourhood of $\mathcal{M}$ conforms to the transition relations. The run is accepting, as the c-loc mappings of the maximal events agree on the graph $G_{\cancel{\mathcal{M}}}$, and the component-numbers as well as the F-sets match the graph $G_{\cancel{\mathcal{M}}}$ perfectly. □

9.4 Decision Procedures and Discussions

The verification problems 21 − 29 are decidable in the case of $\mathsf{UA}(k)$. Since the split-width of $\mathsf{UA}(k)$ is bounded by $k+1$, and since $\mathsf{UA}(k)$ is MSO definable, we could employ the decision procedures studied in Chapter 4.

For recognising $\mathsf{UA}(k)$ over DSTs, we can employ the simulation of the controller $\mathcal{S}_{\mathsf{UA}(k)}$ by a tree-automaton over valid DSTs. There is a tree-automaton $\mathcal{A}^k(\mathcal{S}_{\mathsf{UA}(k)})$ (cf. page. 77) over valid-$k+1$-DSTs recognises the $k+1$-DST encodings of $\mathsf{UA}(k)$. Thus $\mathsf{UA}(k)$ admits a polynomial (which is $k+1$) bound on the split-width via word-like split-terms, and its $k+1$-DST encodings are recognisable by an exponential sized tree-automaton. Hence the complexities of the decision procedures are as stated in Table 4.4.

The class $\mathsf{UA}(k)$ thus provides a good class for the verification purposes. We have elementary decision procedures for verification problems (except those involving MSO). Moreover, we have a finite state controller for this class.

As a corollary, this also gives us a verification technique for communicating finite state machines over acyclic architectures. The reachability problem for such systems has been considered in the literature [LMP08a, MP11] where it is shown to be PSpace-Complete. Note that, we also get decision procedures for various other problems including model checking against temporal and navigational logic specifications.

From the PSpace-hardness of the reachability of communicating finite state machines over acyclic architectures, we may conclude the optimality of our decision procedure. Notice that the split-width is linear in the number of processes in such a case. Thus we may conclude the the PSpace-hardness of CPDS with respect to $\mathsf{UA}(k)$, as well as k-MSCNs, when k is part of the input.

`lower bounds`

Even in the case of cyclic topology $\mathsf{UA}(k)$ provides very interesting classes, which are not existentially bounded. These can encode time bounded computations of Turing machines. It is interesting to see that we have elementary decision procedures for *all* the verification problems (excluding those involving MSO), even without assuming an existential bound on the communication channels.

Chapter 10

Undirected Acyclic with Simple Stack Loops (k), UASL(k)

Allowing loops UA(k) forms a very interesting class of behaviours of communicating finite state machines. However, they are not so interesting in the case of architectures with stacks. Strict acyclicity does not permit the unrestricted behaviours of a single stack. The behaviour of a single stack as such is not the root of undecidability.

Hence, we will now consider classes which are acyclic except for simple self-loops. These self-loops must be due to stacks, since a queue loop immediately renders all the verification problems undecidable. Also, two stack loops on the same context is again Turing powerful, and hence must be avoided.

In such a setting, simultaneous reads from different contexts must be disallowed for the sake of decidability. Otherwise it has the power to verify the intersection of two context-free languages (cf. Remark 9.4).

In this chapter, we consider a class with the strict restriction of undirected acyclicity, and in the next chapter we will see another class based on more lenient directed acyclicity.

Contents

10.1 Definition and Examples

An MSCN $\mathcal{M}$ belongs to the class Undirected acyclic with simple stack loops (k) (abbr. $\mathsf{UASL}(k)$) if it has a context-decomposition $\mathcal{A}\!\!\!/\!\!\mathcal{M}$ such that $G_{\mathcal{A}\!\!\!/\!\!\mathcal{M}} \in \mathcal{G}_{\mathsf{UASL}(k)}$. That is, $\mathsf{UASL}(k) = \mathcal{G}_{\mathsf{UASL}(k)}$-$\mathbb{MSCN}$. $\mathsf{UASL}(k)$

Recall that simultaneous unrestricted execution of two stacks, or a queue, in a single context is forbidden by Condition UASL (cf. Section 8.2.3). Thus a context can either

- read from a stack and write to all data-structures, or
- read from a queue and write to all *other* data-structures.

The Examples 9.2 and 9.3 for the class $\mathsf{UA}(k)$ are also examples for $\mathsf{UASL}(k)$. However, Example 9.1 is not an example of $\mathsf{UASL}(k)$ as it may have unbounded number of interleaved reads from different data-structures.

Example 10.1. Consider communicating pushdown machines over a tree-like architecture with stacks only on processes without incoming queues. All the MSCNs generated by such systems belong to the class $\mathsf{UASL}(k)$.

Note that Example 10.1 is not an example for $\mathsf{UA}(k)$. Thus the classes $\mathsf{UA}(k)$ and $\mathsf{UASL}(k)$ are orthogonal.

Example 10.2. The context-bounded model checking of multi-pushdown systems was introduced and studied in [QR05], where a context allowed access to only one stack. This under-approximation technique assumes a bound on the number of context switches. The behaviours of multi-pushdown systems with at most $k-1$ context switches are $\mathsf{UASL}(k)$; a context-decomposition as per the contexts of [QR05] confirms the membership in $\mathsf{UASL}(k)$.

10.2 MSO Definability and Bounding Split-width

As in the case of $\mathsf{UA}(k)$, this class is also EMSO definable and admits a bound on split-width.

Lemma 10.3. *The class $\mathsf{UASL}(k)$ is EMSO definable.*

Proof. The formula uses existentially quantified second-order variables to represent each context. The specific requirements of $\mathcal{G}_{\mathsf{UASL}(k)}$ can then be stated easily. □

Theorem 10.4. *If an MSCN $\mathcal{M}$ is in $\mathsf{UASL}(k)$, then the split-width of $\mathcal{M}$ is at most $2k-1$.*

The proof is based on the following lemma.

Lemma 10.5. *Let $\mathcal{M}$ be a split-MSCN. If $G_{\mathcal{M}} \in \mathcal{G}_{\mathsf{UASL}(k)}$ then there is a split-term $s_{\mathcal{M}}$ with width at most $2k-1$ such that $\mathcal{M} \in [\![s_{\mathcal{M}}]\!]$.*

Proof. The proof is by induction on the size of $\mathcal{M}$.

The base case is when all the components of $\mathcal{M}$ are trivial (that is only one event per component). In this case, $\mathcal{M}$ is in the shuffle of several basic split-MSCNs. The number of components of any sub-term never exceeds k. Hence the split-width of $s_{\mathcal{M}}$ in this case is at most k.

For the inductive case, suppose $\mathcal{M}$ has at least one non-trivial component. We will identify two split-MSCNs $\mathcal{M}_1$ and $\mathcal{M}_2$ such that

1. $\mathcal{M} \in [\![\mathsf{merge}(\ldots \mathsf{merge}(\mathcal{M}_1 \sqcup\!\sqcup \mathcal{M}_2)) \ldots]\!]^1$,
2. $G_{\mathcal{M}_i} \in \mathcal{G}_{\mathsf{UASL}(k)}$ for each $i \in \{1, 2\}$ and
3. the size of each $\mathcal{M}_i$ is strictly smaller than that of $\mathcal{M}$.

By induction, there exist $s_{\mathcal{M}_1}$ and $s_{\mathcal{M}_2}$ each with width at most $2k-1$. Moreover, they have at most k components each. Thus the number of components of any split-MSCN in $[\![s_{\mathcal{M}_1} \sqcup\!\sqcup s_{\mathcal{M}_2}]\!]$ is at most $2k$, and thus the number of elastic edges is at most $2k-1$.

Thus the split-term $s_{\mathcal{M}}$ will be $\mathsf{merge}(\ldots \mathsf{merge}(s_{\mathcal{M}_1} \sqcup\!\sqcup s_{\mathcal{M}_2}) \ldots)$. The width of $s_{\mathcal{M}}$ is at most $2k-1$ (i.e., $\mathsf{swd}(s_{\mathcal{M}}) \leq 2k-1$).

Now we describe how to identify the split-MSCNs $\mathcal{M}_1$ and $\mathcal{M}_2$. We have two cases to consider.

Case 1: Suppose a brink e of $\mathcal{M}$ is not accessing any data-structure. Then we let $\mathcal{M}_1$ be $\mathcal{M}$ without e, and $\mathcal{M}_2$ be e.

Case 2: Suppose two brinks of $\mathcal{M}$, say e_1 and e_2, are linked with a $\rhd$ edge. Then we let $\mathcal{M}_1$ be $e_1 \rhd e_2$ and $\mathcal{M}_2$ be $\mathcal{M}$ without $e_1 \rhd e_2$. Note that both $G_{\mathcal{M}_1}$ and $G_{\mathcal{M}_2}$ belong to $\mathcal{G}_{\mathsf{UASL}(k)}$.

Case 3: Suppose $\mathcal{M}$ does not have two brinks connected by a $\rhd$ edge. Let $x_1, \ldots, x_k$ be a topological sorting[2] of the components of $\mathcal{M}$. Let x be the first non-trivial component of $\mathcal{M}$ according to the topological sorting. Let e_1 be the first event of x.

Claim 10.6. *We have that e_1 is a write-event.*

Proof. Suppose e_1 was a read-event. Let e_2 be the matching write-event. e_2 is before e_1, and in the topological sorting it forms a trivial component. Hence e_2 as well as e_1 are brinks linked with $\rhd$ edge. Hence this contradicts the assumption that $\mathcal{M}$ does not have two brinks connected by a $\rhd$ edge. This concludes the proof of Claim 10.6. $\square$

[1] The number of preceding merge operations is $\mathsf{elasticity}(\mathcal{M}_1 \sqcup\!\sqcup \mathcal{M}_2) - \mathsf{elasticity}(\mathcal{M})$.

[2] Since $G_{\mathcal{M}}$ is undirected acyclic, it is also directed acyclic. Hence there exists a topological sorting respecting the direction of the edges.

If e_1 is writing to a queue d, let $x^1 = e_1$ and x^2 be the rest of x such that $x = x^1x^2$. Notice that x^2 is non-empty as x is non-trivial. Also notice that queue loops are disallowed by UASL.

If e_1 is writing to a stack d, we have two cases to consider depending on whether e_2 is in x or not. If e_2 is not in x, we let $x^1 = e_1$ and x^2 be the rest of x such that $x = x^1x^2$. Notice that x^2 is non-empty as x is non-trivial. Again there are no $\rhd$ edges from x^1 to x^2 as the $\rhd$ partner of x^1 is not in x. If e_2 is in x, let x^1 be the prefix of x up to and including e_2, and let x^2 be the rest. Since e_2 is not a brink, x^2 is again non-empty. In this case also there are no $\rhd$ edges from x^1 to x^2. Because of SR, all the $\rhd$ edges within x must be from the stack d itself. Hence if there if a $\rhd$ edge from x^1 to x^2, then together with $e_1 \rhd e_2$, it will violate the LIFO policy on stack d.

We now propagate the splitting induced by that of x to the other components. For this, we view $G_{\mathcal{M}}$ as a tree[3] (or a forest) rooted at the component x. This is always possible, as $G_{\mathcal{M}}$ is undirected acyclic. We propagate the splitting top-to-bottom in this tree. During the propagation, we maintain the following two invariants.

INV1 At every step of the propagation we make sure that there are no $\rhd$ edges from a component with a superscript b to another component with a superscript $3 - b$, for $b \in \{1, 2\}$.

$y \rhd z$

We write $y \rhd z$ if there is an event $e_1 \in x$ and $e_2 \in y$ such that $e_1 \rhd e_2$.

INV2 If a node y ($\neq x$) is split into two non-empty components y^1 and y^2 by the propagation, then, letting $\mathsf{father}(y) = z$,

(a) $z \rhd y$.
(b) $z = z^1z^2$ or $z = z^2z^1$ for z^1 and z^2 non-empty.
(c) $z^1 \rhd y^1$ and $z^2 \rhd y^2$.

For each component y (other than x) taken in a top-to-bottom order (for example in a breadth/depth-first manner), we do the following: Let z be the parent of y (in the tree).

- If $z = z^1$, then $y^1 = y$ and y^2 is undefined.
- If $z = z^2$, then $y^2 = y$ and y^1 is undefined.
- If $z = z^1z^2$, we consider the following sub-cases.
  * If $y \rhd z$: From INV2, $\mathsf{father}(z) \rhd z$. From SR, z can read only one data-structure, call it d. Hence $(y, z) \in E_d$ and $(\mathsf{father}(z), z) \in E_d$. Hence both y and $\mathsf{father}(z)$ belong to the same process $\mathsf{Writer}(d)$ and are ordered. From INV2-(c), it follows that y is not connected to both z^1 and z^2. We have four cases to consider, as described in the following table.

[3]This tree is not directed. In this tree every node y has at most one parent z, but the direction of the $\rhd$ edge could be either from x to y or from y to z.

$y \rhd z$	d is a queue	d is a stack
$y < \mathsf{father}(z)$	$y^1 = y \not\rhd z^2$ and y^2 is undefined	$y^2 = y \not\rhd z^1$ and y^1 is undefined
$\mathsf{father}(z) < y$	$y^2 = y \not\rhd z^1$ and y^1 is undefined	$y^1 = y \not\rhd z^2$ and y^2 is undefined

The invariants INV1 and INV2 are trivially true in this case.

* If $z \rhd y$: Let d be the data-structure being read in y. We split y as described in the following table.

d is a queue	d is a stack
Let y^1 be the shortest prefix of y to contain all the $\rhd$ edges from z^1, and let y^2 be the rest such that $y = y^1 y^2$.	Let y^1 be the shortest suffix of y to contain all the $\rhd$ edges from z^1, and let y^2 be the rest such that $y = y^2 y^1$.

Notice that in both cases we have $y^2 \not\rhd z^1 \rhd y^1$. Moreover, thanks to SR and the access policy on d (LIFO or FIFO), there cannot be any $\rhd$ edge between y^1 and y^2. Thus the invariants INV1 and INV2 are satisfied in this case.

– If $z = z^2 z^1$: This case is symmetric to the case before.

For each context y not treated by the above top-to-bottom ordering of the tree (these are the components which are not in the same connected component of x in $G_{\mathcal{M}}$), we let $y^1 = y$ and leave y^2 undefined.

Thus the above procedure gives us a way to split the components into two. All those[4] with a superscript i forms the split-MSCN $\mathcal{M}_i$. Observe that there are no $\rhd$ edge between components of $\mathcal{M}_i$ and those of $\mathcal{M}_{3-i}$.

This gives us the required split-MSCNs $\mathcal{M}_1$ and $\mathcal{M}_2$. Notice that $\mathcal{M}_i \in \mathsf{UASL}(k)$. Clearly, $\mathcal{M} \in \mathsf{merge}(\ldots \mathsf{merge}(s_{\mathcal{M}_1} \sqcup s_{\mathcal{M}_2}) \ldots)$. This concludes the proof of Lemma 10.5. $\square$

Proof of Theorem 10.4. Let $\mathcal{M}$ be a context-decomposition of $\mathcal{M}$ such that $G_{\mathcal{M}} \in \mathcal{G}_{\mathsf{UASL}(k)}$. $\mathcal{M} \in \mathsf{merge}(\ldots \mathsf{merge}(s_{\mathcal{M}}) \ldots)$, and the width of $s_{\mathcal{M}}$ is at most $2k-1$, thanks to Lemma 10.5. Thus the split-width of $\mathcal{M}$ is at most $2k-1$. $\square$

10.3 Controller

The controller is a particular case of the controller defined for generic class $\mathcal{G}$ of finite context-graphs in Section 9.3 with a uniform replacement of $\mathbb{MSCN}_{\mathcal{G}}$ with $\mathsf{UASL}(k)$ and $\mathcal{G}$ with $\mathcal{G}_{\mathsf{UASL}(k)}$.

More explicitly, the context-graphs appearing in the control locations are those belonging to $\mathcal{G}_{\mathsf{UASL}(k)}$. Thus the controller will initially guess a graph from $\mathcal{G}_{\mathsf{UASL}(k)}$ and try to verify it along the run. We denote the controller for $\mathsf{UASL}(k)$ by $\mathcal{S}_{\mathsf{UASL}(k)}$. $\mathcal{S}_{\mathsf{UASL}(k)}$ is a sound and complete controller for $\mathsf{UASL}(k)$.

[4]We may discard the empty components.

10.4 Decision Procedures and Discussions

The verification problems 21 – 29 are for $\mathsf{UASL}(k)$ as well. Since $\mathsf{UASL}(k)$ is MSO definable and since the split-width is bounded by $2k-1$, we again employ the decision procedures studied in Chapter 4.

The reasoning is similar to the case of $\mathsf{UA}(k)$. There is a tree-automaton $\mathcal{A}^k(\mathcal{S}_{\mathsf{UASL}(k)})$ over $(2k-1)$-DST-Labels which recognises embeddings of MSCNs from $\mathsf{UASL}(k)$. The size of $\mathcal{A}^k(\mathcal{S}_{\mathsf{UASL}(k)})$ (cf. page. 77) is exponential in k. Thus the class $\mathsf{UASL}(k)$ admits a polynomial bound on split-width and an exponential sized tree-automaton recognising its DST-encodings. Hence, the verification problems follow the complexities dictated in Table 4.3.

Thus to conclude, the class $\mathsf{UASL}(k)$ also proves to be a good class for under-approximate verification of CPDS. It allows us to have elementary decision procedures for the verification problem (except those involving MSO) and is also efficiently implementable.

This class compares well with other classes studied in the literature. In [LMP08a, HLMS10], the authors have studied the emptiness problem of communicating pushdown systems over an acyclic topology. They obtain decidability when a restriction called 'well-queuing' is imposed. This restriction requires the local stack to be empty when a process accesses a queue. The class $\mathsf{UASL}(k)$ is orthogonal to the well-queuing restriction. The latter permits an unbounded interleaving of reads from a queues and stacks, whereas the former permits reading from a queue even when the local stack is not empty. The class $\mathsf{UASL}(k)$ also allows for the verification of cyclic architectures.

Chapter 11

Directed Acyclic with Simple Stack Loops (k), $\mathsf{DASL}(k)$

Contents

11.1 Definition and Examples

An MSCN $\mathcal{M}$ belongs to the class $\mathsf{DASL}(k)$ if it has a context-decomposition $\mathcal{M}$ such that $G_{\mathcal{M}} \in \mathcal{G}_{\mathsf{DASL}(k)}$. That is, $\mathsf{DASL}(k) = \mathcal{G}_{\mathsf{DASL}(k)}$-$\mathbb{MSCN}$. $\mathsf{DASL}(k)$

This class is a generalisation of $\mathsf{UASL}(k)$ by relaxing the strong undirected acyclicity requirement by weaker directed acyclicity requirement. Like in the case of $\mathsf{UASL}(k)$, a context of $\mathsf{DASL}(k)$ can either

- read from a stack and write to *all* data-structures, or
- read from a queue and write to *all other* data-structures.

The Examples 9.2, 9.3, 10.1 and 10.2 are also examples for $\mathsf{DASL}(k)$.

Example 11.1. Consider communicating pushdown machines over a directed acyclic architecture with stacks only on roots. All the MSCNs generated by such systems belong to the class $\mathsf{DASL}(k)$.

Example 11.2. The well-studied bounded phase restriction [LMP07] on multi-pushdown systems is an example of DASL. A phase allows pops (or reads) from at most one stack, while pushes (writes) to all stacks are permitted. Each phase is also a DASL context. Thus bounded number of phases implies bounded number of contexts. Since multi-pushdown systems have only one process, no cycles other than loops are possible.

11.2 MSO Definability and Bounding Split-width

$\mathsf{DASL}(k)$ is also EMSO definable and admits a bound on split-width.

Lemma 11.3. *The class* $\mathsf{DASL}(k)$ *is* EMSO *definable.*

Proof. As in the case of previous classes, we employ second order variables to identify the context decomposition. Directed acyclicity can then be stated in first-order as there are at most k contexts. □

Theorem 11.4. *If an MSCN* $\mathcal{M} \in \mathsf{DASL}(k)$*, then the split-width of* $\mathcal{M}$ *is at most* $2^k - 1$.

Proof. The proof proceeds in two steps. In the first step we give a reduction from $\mathsf{DASL}(k)$ to a stronger version of DASL in which every component has in-degree at most one. This causes an exponential blow-up. Then, in the second step we show that any MSCN belonging to the stronger version of DASL has linear sized split-width.

Stronger version of DASL: $\mathsf{sDASL}(k)$. An MSCN $\mathcal{M}$ is in $\mathsf{sDASL}(k)$ if it admits a context-decomposition $\mathcal{M}$ such that $G_{\mathcal{M}} \in \mathcal{G}_{\mathsf{DASL}(k)}$ and in addition, the in-degree of every component is at most one, excluding self-loops. We denote the set of such context-graphs by $\mathcal{G}_{\mathsf{sDASL}(k)}$.

$\mathcal{G}_{\mathsf{sDASL}(k)}$

A context graph G belongs to the class $\mathcal{G}_{\mathsf{sDASL}(k)}$ if

DASL G is directed acyclic with simple stack self-loops,

B G has at most k vertices: $|V| \leq k$, and

SSR for all contexts x, y, z and data-structures d and d',
if $(y, x) \in E_d$ and $(z, x) \in E_{d'}$, then $d = d'$, and $z = x$ or $z = y$.

Example 11.5. Some (counter-)examples of $\mathcal{G}_{\mathsf{sDASL}(4)}$, $\mathcal{G}_{\mathsf{UASL}(4)}$ and $\mathcal{G}_{\mathsf{DASL}(4)}$ are given in Table 11.1.

Lemma 11.6. *If* $\mathcal{M} \in \mathsf{DASL}(k)$*, then* $\mathcal{M} \in \mathsf{sDASL}(2^{k-1})$.

Proof. Consider the context decomposition witnessing membership in $\mathsf{DASL}(k)$. Call it $\mathcal{M}'$. $G_{\mathcal{M}'}$ has at most k vertices. Let $x_1, \ldots, x_k$ be a topological ordering of the components of $\mathcal{M}'$. It follows from Condition SR that a component x_i may have incoming $\rhd$ edges from any preceding component, but all these must be from the same data-structure.

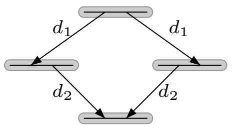
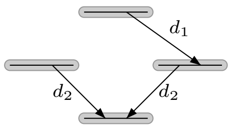
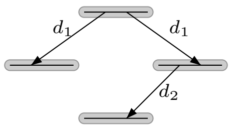
			
$\mathcal{G}_{\mathsf{sDASL}(4)}$	✗	✗	✓
$\mathcal{G}_{\mathsf{UASL}(4)}$	✗	✓	✓
$\mathcal{G}_{\mathsf{DASL}(4)}$	✓	✓	✓

Table 11.1: $\mathcal{G}_{\mathsf{sDASL}(4)}$, $\mathcal{G}_{\mathsf{UASL}(4)}$ and $\mathcal{G}_{\mathsf{DASL}(4)}$ yes/no.

We split each component x_i into $f(i)$ many new components (some of them may be empty). Call them $x_{i,1}, \dots, x_{i,f(i)}$. Each of the new component $x_{i,j}$ has incoming $\rhd$ edges from at most one component other than itself.

Here $f(i)$ is a number which is a function of i. For $i = 1$, $f(i) = 1$. For $i > 1$, $f(i) = 2^{i-2}$. This estimate can be verified. Notice that the number of new components preceding x_i in this procedure is

$$f(1) + \sum_{j=2}^{i-1} f(j) = 1 + \sum_{j=2}^{i-1} 2^{j-2} = 1 + 2^{i-2} - 1 = 2^{i-2}.$$

Thus a component needs to be split into at most $f(i)$ many new ones.

This splitting is possible, because all the reads are from the same data-structure. We will see the splitting more closely now.

Suppose component x_i has incoming edges from E_d where d is a queue. Then, $x_{i,1}$ is a factor of x_i which contains all the $\rhd$ edges from $x_{1,1}$; $x_{i,2}$ is a factor of x_i which contains all the $\rhd$ edges from $x_{2,1}$; etc., and $x_{i,f(i)}$ is a factor of x_i which contains all the $\rhd$ edges from $x_{i-1,f(i-1)}$. Further we require that $x_i = x_{i,1} \cdot x_{i,2} \dots \cdot x_{i,f(i)}$.

Suppose component x_i has incoming edges from E_d where d is a stack. Due to the LIFO policy on stacks, the new components will be attached to the preceding ones in the reverse order. That is, $x_{i,1}$ is a factor of x_i which contains all the $\rhd$ edges from $x_{i-1,f(i-1)}$, $x_{i,2}$ is a factor of x_i which contains all the $\rhd$ edges from $x_{i-1,f(i-1)-1}$, etc. and $x_{i,f(i)}$ contains all the $\rhd$ edges from $x_{1,1}$. As before $x_i = x_{i,1} \cdot x_{i,2} \dots \cdot x_{i,f(i)}$. We may also assume the the factors $x_{i,1}$, $\dots x_{i,f(i)-1}$, if non-empty, end with a read-event. This forbids $\rhd$ edges between $x_{i,1}, \dots x_{i,f(i)}$.

The required split-MSCN $\mathcal{M}$ is the one with

$$x_{1,1}, x_{2,1}, \dots, x_{i,1}, \dots, x_{i,f(i)}, \dots, x_{k,f(k)}$$

as its components.

The total number of components is at most $f(k+1) = 2^{k-1}$. □

The second part of the proof is showing the linear bound on split-width for $\mathsf{sDASL}(k)$.

Lemma 11.7. *If $\mathcal{M} \in \mathsf{sDASL}(k)$, then the split-width of $\mathcal{M}$ is at most $2k-1$.*

The proof of this lemma is based on the following lemma.

Lemma 11.8. *Let $\cancel{\mathcal{M}}$ be a split-MSCN. If $G_{\cancel{\mathcal{M}}} \in \mathcal{G}_{\mathsf{sDASL}(k)}$ then there is a split-term $s_{\cancel{\mathcal{M}}}$ with width at most $2k-1$ such that $\cancel{\mathcal{M}} \in [\![s_{\cancel{\mathcal{M}}}]\!]$.*

Proof. The proof is by induction on the size of $\cancel{\mathcal{M}}$.

The base case is when all the components of $\cancel{\mathcal{M}}$ are trivial (that is only one event per component). In this case, $\cancel{\mathcal{M}}$ is in the shuffle of several basic split-MSCNs. The number of components of any subterm never exceeds k. Hence the split-width in this case is at most $k-1$.

For the inductive case, suppose $\cancel{\mathcal{M}}$ has at least one non-trivial component. We will identify two split-MSCNs $\cancel{\mathcal{M}}_1$ and $\cancel{\mathcal{M}}_2$ such that both $G_{\cancel{\mathcal{M}}_1} \in \mathcal{G}_{\mathsf{sDASL}(k)}$ and $G_{\cancel{\mathcal{M}}_2} \in \mathcal{G}_{\mathsf{sDASL}(k)}$ and the size of $\cancel{\mathcal{M}}_i$ is strictly smaller than that of $\cancel{\mathcal{M}}$. By induction, there exist $s_{\cancel{\mathcal{M}}_1}$ and $s_{\cancel{\mathcal{M}}_2}$ with width at most $2k-1$. The split-expression $s_{\cancel{\mathcal{M}}} = \mathsf{merge}(\mathsf{merge}(\ldots \mathsf{merge}(s_{\cancel{\mathcal{M}}_1} \sqcup\!\sqcup\, s_{\cancel{\mathcal{M}}_2})\ldots))$. Since both $\cancel{\mathcal{M}}_1$ and $\cancel{\mathcal{M}}_2$ have at most k components each, the total number of components at the shuffle node is at most $2k$. Hence the width of $s_{\cancel{\mathcal{M}}}$ is at most $2k-1$.

Now we describe how to identify the split-MSCNs $\cancel{\mathcal{M}}_1$ and $\cancel{\mathcal{M}}_2$. We have two cases to consider.

Case 1: Suppose a brink e of $\cancel{\mathcal{M}}$ is not accessing any data-structure. Then we let $\cancel{\mathcal{M}}_1$ be $\cancel{\mathcal{M}}$ without e, and $\cancel{\mathcal{M}}_2$ be e.

Case 2: Suppose two brinks of $\cancel{\mathcal{M}}$, say e_1 and e_2, are linked with a $\rhd$ edge. Then we let $\cancel{\mathcal{M}}_1$ be $e_1 \rhd e_2$ and $\cancel{\mathcal{M}}_2$ be $\cancel{\mathcal{M}}$ without $e_1 \rhd e_2$. Note that both $G_{\cancel{\mathcal{M}}_1}$ and $G_{\cancel{\mathcal{M}}_2}$ belong to $\mathcal{G}_{\mathsf{sDASL}(k)}$. Indeed, $\cancel{\mathcal{M}} \in \mathsf{merge}(\mathsf{merge}(s_{\cancel{\mathcal{M}}_1} \sqcup\!\sqcup\, s_{\cancel{\mathcal{M}}_2}))$.

Case 3: Suppose $\cancel{\mathcal{M}}$ does not have two brinks connected by a $\rhd$ edge. Let $x_1, \ldots, x_k$ be a topological sorting of the components of $\cancel{\mathcal{M}}$. Let x_i be the first non-trivial component. We split the component x_i into two non-empty components x_i^1 and x_i^2 such that $x_i = x_i^1 \cdot x_i^2$ and there are no edges from x_i^1 to x_i^2. We explain below why such a splitting is possible whether x_i has self-loops or not.

> If x_i does not have self-loops, any non-trivial splitting suffices. If x_i has a self-loop, let d be the stack being read in x_i. Consider the first event (brink) of x_i. Let us call it e_1. The event e_1 alone forms x_i^1 if either e_1 is not taking part in a $\rhd$ edge, or the $\rhd$ neighbour of e_1 is not in x_i (that is, $e_2 \rhd e_1$ or $e_1 \rhd e_2$, and $e_2 \notin x_i$). Otherwise, e_1 has a $\rhd$ neighbour e_2 which also belongs to x_i ($e_1 \rhd e_2$ and $e_2 \in x_i$). The event e_2 is not a brink, as otherwise this would have been in Case 1. We let x_i^1 to be the prefix of x_i up to and including e_2. Indeed x_i^2 is the rest of x_i. There are no $\rhd$ edges linking x_i^1 and x_i^2 in all these cases.

We now propogate the splitting induced by that of x_i to the other components.

For $j < i$:
If x_i^b has an incoming edge from x_j then $x_j^b = x_j$ for $b \in \{1,2\}$. Since $j < i$ and x_j is singleton. Let $x_j^1 = x_j$ for all other $j < i$.

For $j > i$:
For each $j > i$ taken in the order, we do the following: If x_j has no incoming edges from any previous component, we let $x_j^1 = x_j$. If x_j has incoming edges from a previous component $x_{j'}$ (note that there is only one such component):

- If $x_{j'}^2$ is undefined, then let $x_j^1 = x_j$.
- If $x_{j'} = x_{j'}^1 \cdot x_{j'}^2$:
    * If x_j has reads from a queue data-structure (that is, $(x_{j'}, x_j) \in E_d$ for $d \in$ **Queues**): $x_j = x_j^1 \cdot x_j^2$ such that all the $\rhd$ edges from $x_{j'}^b$ are incident on x_j^b for $b \in \{1,2\}$.
    * If x_j has reads from a stack data-structure (that is, $(x_{j'}, x_j) \in E_d$ for $d \in$ **Stacks**): $x_j = x_j^2 \cdot x_j^1$ such that all the edges from $x_{j'}^b$ are incident on x_j^{3-b} for $b \in \{1,2\}$. Moreover, we ensure that there are no $\rhd$ edges between x_j^2 and x_j^1 by letting x_j^2 to be minimal. Notice that the order of the indices is reversed in order to respect the LIFO policy on stacks.
- If $x_{j'} = x_{j'}^2 \cdot x_{j'}^1$:
    * If x_j has reads from a queue data-structure (that is, $(x_{j'}, x_j) \in E_d$ for $d \in$ **Queues**): $x_j = x_j^2 \cdot x_j^1$ such that all the $\rhd$ edges from $x_{j'}^b$ are incident on x_j^b for $b \in \{1,2\}$.
    * If x_j has reads from a stack data-structure (that is, $(x_{j'}, x_j) \in E_d$ for $d \in$ **Stacks**): $x_j = x_j^1 \cdot x_j^2$ such that all the edges from $x_{j'}^b$ are incident on x_j^{3-b} for $b \in \{1,2\}$. Moreover, we ensure that there are no $\rhd$ edges between x_j^1 and x_j^2 by choosing x_j^1 minimal. Notice that the order of the indices is reversed in order to respect the LIFO policy on stacks.

Finally, $\mathcal{M}_b$ is the split-MSCN with x_j^b as its components. Observe that there are no $\rhd$ edge between components of $\mathcal{M}_b$ and those of $\mathcal{M}_{3-b}$. Clearly, $\mathcal{M} \in \mathsf{merge}(\ldots\mathsf{merge}(s_{\mathcal{M}_1} \sqcup s_{\mathcal{M}_2})\ldots)$. Notice that for each b, $G_{\mathcal{M}_b}$ is $\mathsf{sDASL}(k)$. $\square$

Proof of Lemma 11.7. Let $\mathcal{M}$ be a $\mathsf{sDASL}(k)$ context-decomposition of $\mathcal{M}$. From Lemma 11.8 there exists $s_{\mathcal{M}}$ of width at most $2k-1$ such that $\mathcal{M} \in [\![s_{\mathcal{M}}]\!]$. We have $\mathcal{M} \in [\![s_{\mathcal{M}}]\!]$ where $s_{\mathcal{M}} = \mathsf{merge}(\ldots\mathsf{merge}(s_{\mathcal{M}})\ldots)$ and this proves the bound on split-width. $\square$

Theorem 11.4 follows from Lemma 11.6 and Lemma 11.7. $\square$

11.3 Controller

The controller for $\mathsf{DASL}(k)$ is a particular instantiation of $\mathcal{S}_{\mathsf{DASL}(k)}$ defined in Section 9.3. We call this controller $\mathcal{S}_{\mathsf{DASL}(k)}$. The context-graphs appearing in the control locations are those belonging to $\mathcal{G}_{\mathsf{DASL}(k)}$. The controller will initially guess a graph from $\mathcal{G}_{\mathsf{DASL}(k)}$ and try to verify it along the run. The controller $\mathcal{S}_{\mathsf{DASL}(k)}$ is sound and complete for $\mathsf{DASL}(k)$.

11.4 Decision Procedures and Discussions

The class $\mathsf{DASL}(k)$ also offers decidability for the verification problems, similar to $\mathsf{UA}(k)$ and $\mathsf{UASL}(k)$. However, since the split-width is bounded by $2^k - 1$, we will be using tree-automata over (2^{k-1})-DST-Labels. As before the tree-automaton $\mathcal{A}^k(\mathcal{S}_{\mathsf{DASL}(k)})$ recognised the encoding of $\mathsf{DASL}(k)$ among valid (2^{k-1})-DSTs.

Since there is an exponential bound on the split-width, and since there is a double exponential sized tree-automaton over valid DSTs recognising the encodings of $\mathsf{DASL}(k)$, the complexities of the decision procedure are as stated in Table 4.5.

Notice that, our decision procedure gives a double exponential time decision procedure for the emptiness of k-phase multi-pushdown systems, which is known to be 2-ExpTime-Complete [LMP07, LMP08b]. Thus our bound on split-width is asymptotically optimal, as well as the decision procedure. Satisfiability and model checking of temporal logics and PDL over bounded phase multiply-nested words have been addressed in [BCGZ11], but only when k is not part of the input. Hence the complexity upper bounds of these problems when k is part of the input are new.

Notice that we cover MSCNs from generic architectures and not just multi-pushdown systems. These architectures could be cyclic as well. We cover behaviours which are not necessarily existentially bounded.

In the case of communicating finite state machines, the MSCNs in $\mathsf{DASL}(k)$ also admits a linearisation with at most k contexts, and vice versa. The reachability problem in this case has been addressed in [LMP08a] under the name bounded context-switching reachability problem for non-recursive queuing concurrent programs. There, it is shown to be decidable in time double exponential in the number of contexts and exponential in the size of the CPDS. Our decision procedure improves this upper bound to double exponential in the number of contexts, but only polynomial in the size of the CPDS. Also note that, our decision procedure can be applied even in the case of recursive queuing programs.

Chapter 12

Bounded Scope (k)

Contents

The class bounded scope (k) is orthogonal to all the previous classes. It does not bound the size of the context-graph. But, the contexts are more restrictive.

12.1 Definition and Examples

An MSCN $\mathcal{M}$ belongs to the class $\mathsf{BS}(k)$ if it has a context-decomposition $\mathcal{A\!\!\!/}$ such that $G_{\mathcal{A\!\!\!/}} \in \mathcal{G}_{\mathsf{BS}(k)}$. That is, $\mathsf{BS}(k) = \mathcal{G}_{\mathsf{BS}(k)}$-$\mathbb{MSCN}$. $\mathsf{BS}(k)$

Notice that the condition SRW disallows simultaneous writes and reads on a queue data-structure, or self-queues. Thus the contexts are essentially

- accessing a single stack in either direction, but no other data-structure, or
- reading from a queue but no other data-structure access, or
- writing to a queue but no other data-structure access.

This class imposes a bound on the number of contexts between a write and the corresponding read.

Example 12.1. Consider Example 9.3 which simulates k-computation steps of a Turing machine. It is not an example of $\mathsf{BS}(k)$. However, the dual restriction, which puts a limit on the tape length but allows unboundedly many steps is indeed bounded scope. A trivial context-decomposition in which each context is a singleton witnesses its membership in $\mathsf{BS}(k)$.

Example 12.2. The above example gives rise to an existentially bounded[1] MSCN language. Example 9.2 is also an example of $\mathsf{BS}(k)$ which is not existentially bounded. But Example 9.2 has only 4 contexts. However, an unbounded repetition of the pattern (Figure 12.1) in Example 9.2 is also an example MSCN in the class $\mathsf{BS}(4)$ which has unbounded number of contexts and which is not existentially bounded.

Example 12.3. The class $\mathsf{BS}(k)$ in case of multi-pushdown systems is essentially the scope bounded multi-pushdown systems. The latter has been introduced in [LN11] and is a trending topic of current research [LN12, LP12].

12.2 MSO Definability and Bounding Split-width

Proposition 12.4. *The class* $\mathsf{BS}(k)$ *is* MSO *definable.*

Proof. Instead of verifying conditions SRW, DASL and BS, we will equivalent conditions SRW, SL and BS (thanks to Remark 8.6). The conditions SRW and SL are easy to enforce on any guessed context. The difficulty is to ensure BS.

The idea is to forbid the existence of $\rhd$ edges which span over more than k contexts. Thus the formula will negate the existence of a $\rhd$ edge with $k+1$ contexts in between. To identify k contexts in between a matching write and read, it is sufficient to identify $k+1$ events accessing data-structures, such that consecutive events from this set accesses different data-structures. This is in order to force a change of context.

Consider the following formula:

$$
\begin{aligned}
&\neg\exists x_0\,\exists x_1\cdots\exists x_k\,\exists x'_1\,\cdots\,\exists x'_{k-1}\,\exists y_0\,\exists y_1\,\cdots\,\exists y_{k-1}\,\exists y'_1\,\cdots\,\exists y'_{k-1}\\
&\quad x_0 \rhd x_k \wedge \bigwedge_{i\in\{1,\dots,k-1\}} (x_i \rhd x'_i) \vee (x'_i \rhd x_i) \wedge \bigwedge_{i\in\{0,\dots,k-1\}} (y_i \rhd y'_i) \vee (y'_i \rhd y_i)\\
&\qquad \wedge \bigwedge_{i\in\{1,\dots,k-1\}} x_i < x_k \wedge \neg(x_i < x_0)\\
&\qquad \wedge \bigwedge_{i\neq j\in\{0,\dots k\}} \Big[x_i \neq x_j\\
&\qquad\qquad \wedge \bigwedge_{d\in\mathbf{Stacks}} \big(d(x_i)\wedge d(x_j)\wedge x_i < x_j\\
&\qquad\qquad\qquad \implies x_i < y_i < x_j \wedge \neg d(y_i)\big)\\
&\qquad\qquad \wedge \bigwedge_{d\in\mathbf{Queues}} \big(d(x_i)\wedge d(x_j)\wedge x_i \rhd x'_i \wedge x_j \rhd x'_j \wedge x_i < x_j\\
&\qquad\qquad\qquad \implies x_i < y_i < x_j \wedge (\neg d(y_i) \vee y'_i \rhd y_i)\big)\\
&\qquad\qquad \wedge \bigwedge_{d\in\mathbf{Queues}} \big(d(x_i)\wedge d(x_j)\wedge x'_i \rhd x_i \wedge x'_j \rhd x_j \wedge x_i < x_j\\
&\qquad\qquad\qquad \implies x_i < y_i < x_j \wedge (\neg d(y_i) \vee y_i \rhd y'_i)\big)\Big]
\end{aligned}
$$

[1]An existentially bounded language assumes a bound on the size of the data-structures. An MSCN in the language can be generated by such a size bounded data-structure.

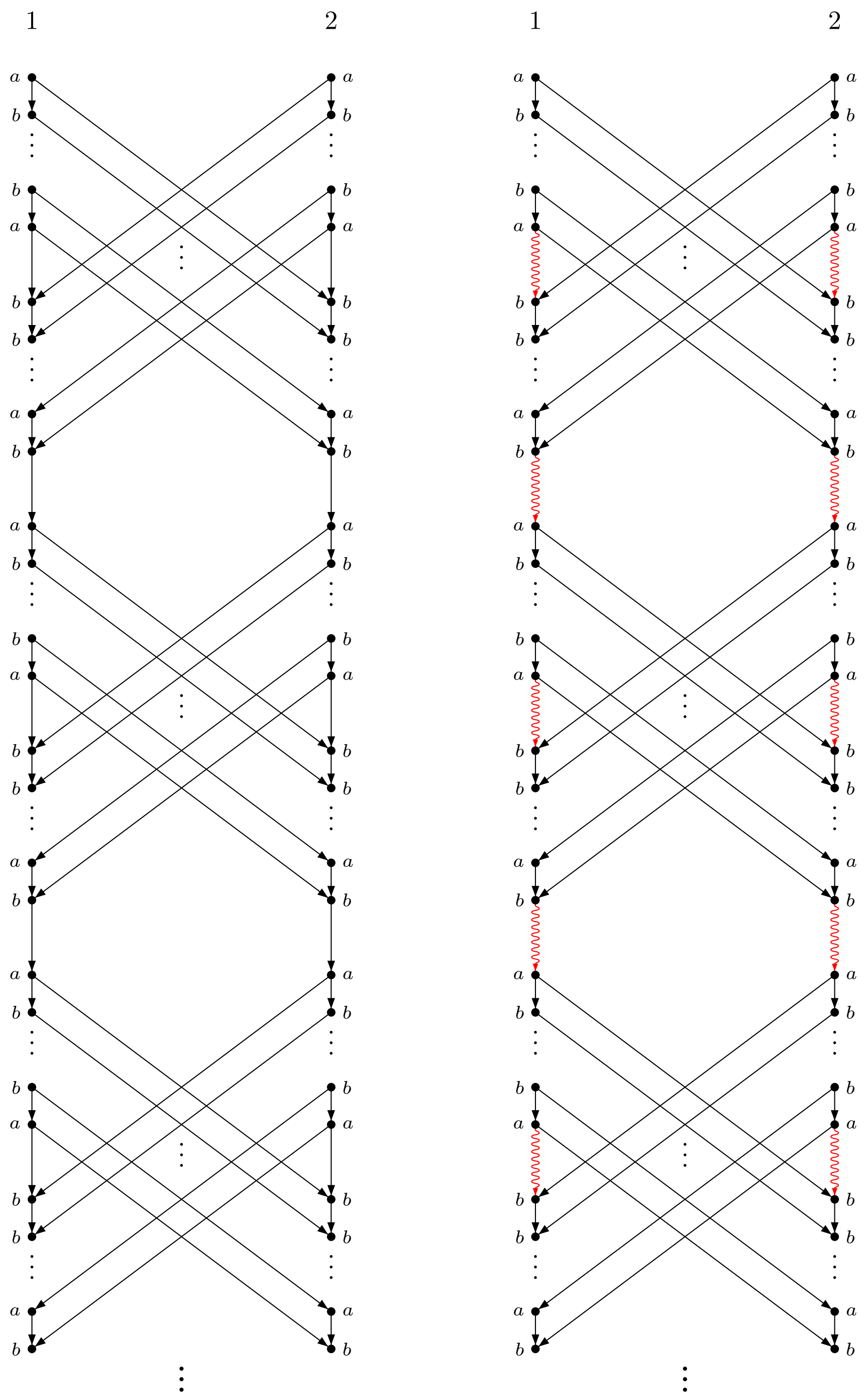


Figure 12.1: An MSCN which is not existentially bounded and a witnessing context-decomposition for its membership in $\mathsf{BS}(4)$.

The formula aims to witness a violation of the scope boundedness. The first order variables x_0 and x_k stand for the events linked by $\rhd$ edge which spans over more than k contexts. The variables $x_1, \ldots, x_{k-1}$ stand for $k-1$ events which are the representatives of $k-1$ contexts which occur in between x_0 and x_k. These $k-1$ events must access data-structures (to emphasise a context) and must be pair-wise distinct. We use the variables $x'_1, \ldots, x'_{k-1}$ to represent their respective $\rhd$-neighbours.

Moreover, in order to force a change of context between any two such events, we require that any two events which could potentially belong to the same context (accessing the same queue in the same direction, or accessing the same stack), must have another event in between them. This new event must be accessing a different data-structure, or accessing the queue in the opposite direction. We use the variable y_i to mark the change of context between x_i and any other x_j accessing the same data-structure at a later point of time (note that, x_i and x_j will be on the same process as they are accessing the same data-structure in the same direction). We use the variable y'_i to indicate the $\rhd$-neighbour of y_i. Notice that those x_i which do not satisfy the premise of the implication, (like those which represent the last context on a process) do not need a y_i, and in this case y_i could be same as x_i itself.

Notice that, even if the source and the target are separated by more than $k+1$ contexts, the above formula would detect it. Any $k-1$ contexts which occur in between need to be witnessed in addition to the source and the target contexts. $\square$

Remark 12.5. The above formula is first-order if we allow the partial order relation $<$ in the syntax. Otherwise, expressing the partial order relation, which is the transitive closure of the $\rightarrow$ and $\rhd$ edges, require second order quantification. This can be noticed in our definition of $<$ as a macro in Section 2.5.

Let $\mathsf{FO}(<)$ denote the first-order logic supplemented with the partial order relation. It follows that:

Proposition 12.6. *The class* $\mathsf{BS}(k)$ *is* $\mathsf{FO}(<)$ *definable.*

The class $\mathsf{BS}(k)$ also admits a bound on split-width.

Theorem 12.7. *If an MSCN* $\mathcal{M} \in \mathsf{BS}(k)$*, then the split-width of* $\mathcal{M}$ *is at most* $(\mathfrak{d}+1)k+2$ *where* $\mathfrak{d}$ *is the number of data-structures in the architecture.*

Proof. Let $\mathcal{M}$ be a context-decomposition witnessing membership in $\mathsf{BS}(k)$. That is, $G_{\mathcal{M}} = (V, (E_d)_{d \in \mathbf{DS}}) \in \mathcal{G}_{\mathsf{BS}(k)}$. The proof proceeds in two steps. In the first step we show that there exists a topological sorting $x_1, x_2, \ldots$ of the contexts of $\mathcal{M}$ such that if $(x_i, x_j) \in E_d$, then $j - i < (\mathfrak{d}+1)k$. In the second step we show that if a split-MSCN $\mathcal{M}$ admits a total ordering of the contexts such that $\rhd$ edges have at most m contexts between the source and target with respect to the total order, then the split-width of $\mathcal{M}$ is at most $m+2$.

Lemma 12.8. *Let $\mathcal{M}$ be a split-MSCN in $\mathsf{BS}(k)$. Then there exists a topological sorting $x_1, x_2, \ldots$ of the contexts of $\mathcal{M}$ such that if $e_1 \rhd e_2$ and $e_1 \in x_i$ and $e_2 \in x_j$, then $j - i < (\mathfrak{d} + 1)k$.*

Proof. We will describe how to extract a required ordering. Consider $G_{\mathcal{M}}$. First we show that any prefix of $G_{\mathcal{M}}$ can be augmented by at most k new contexts to match all the pending writes on a particular data-structure.

Let $G_{\mathcal{M}} = (V, (E_d)_{d \in \mathbf{DS}}) \in \mathcal{G}_{\mathsf{BS}(k)}$. We say subset $U \subseteq V$ a *prefix* of $G_{\mathcal{M}}$ if it is down-ward closed with respect to $\preccurlyeq$. Recall that $\preccurlyeq$ is a partial order in the case of $\mathcal{G}_{\mathsf{BS}(k)}$. Prefix U is a *strict* prefix of $G_{\mathcal{M}}$ if $U \subsetneq V$. `prefix`

Lemma 12.9. *Let $G_{\mathcal{M}} = (V, (E_d)_{d \in \mathbf{DS}}) \in \mathcal{G}_{\mathsf{BS}(k)}$ and let U be any strict prefix of $G_{\mathcal{M}}$. For each $d \in \mathbf{DS}$, there exists a bigger prefix U' (i.e., $U \subsetneq U' \subseteq V$) such that*

1. *$|U' \setminus U| \leq k$ and*
2. *all the unmatched writes on data-structure d are in $U' \setminus U$.*
   That is, if $(x, y) \in E_d$ and $x \in U'$ and $y \notin U'$ then $x \notin U$. Equivalently, all the writes to data-structure d in U are matched in U'. That is, if $(x, y) \in E_d$ and $x \in U$, then $y \in U'$.

Proof. If the data-structure d does not have any unmatched write in U, then we augment it with any minimal context from $V \setminus U$ to get U'. If U' contains any unmatched write on d, then clearly it is from this new context and hence not in U.

Suppose the data-structure d has some unmatched writes in U. Let e_1 be the last-to-be-read unmatched write on d in U. If d is a stack, then e_1 is the first unmatched write on d; and if d is a queue, e_1 is the last unmatched write on d. Let e_2 be the matching read. Further let x and y be the respective contexts of e_1 and e_2. That is $e_1 \in x$ and $e_2 \in y$. Indeed $y \in V \setminus U$. We will augment U by adding the context y and the necessary contexts from the past of y (precisely the down-ward closure of y as these are required to make it a prefix). Thus $U' = U \cup \downarrow y$ where $\downarrow y$ denotes the down-ward closure of y, i.e. $\downarrow y = \{x \in V \mid x \preccurlyeq y\}$. By Condition BS (cf. 126), the number of contexts in the past of y, but not in the past of x are at most $k - 1$. Also by choosing e_1 as the last-to-be-read unmatched write, we make sure that all the unmatched writes in U are matched in U'. This concludes the proof of Lemma 12.9. □

We write $U \xrightarrow{d} U'$, if it witnesses the above lemma. $U \xrightarrow{d} U'$

We will now give a sequence of prefixes of $G_{\mathcal{M}}$ such that any linear ordering (or completion to a total order of successive prefixes) of it will provide the topological sorting claimed in Lemma 12.8. For this, let $d_1, \ldots d_{\mathfrak{d}}$ be an arbitrary enumeration (ordering) of the data-structures in $\mathbf{DS}$.

Lemma 12.10. *Consider the sequence which follows the enumeration of the data-structures in a round-robin fashion*

$$\emptyset = U_0 \xrightarrow{d_1} U_1 \xrightarrow{d_2} U_2 \ldots \xrightarrow{d_{\mathfrak{d}}} U_{\mathfrak{d}} \xrightarrow{d_1} U_{\mathfrak{d}+1} \ldots \xrightarrow{d_m} U_n = V.$$

For all $i > 0$ and all data-structures $d \in \mathbf{DS}$, all the unmatched writes on data-structure d in the prefix U_i are in $U_i \setminus U_j$ where j is the largest index smaller than i such that $U_j \xrightarrow{d} U_{j+1}$ appears in the above sequence. If $U_j \xrightarrow{d} U_{j+1}$ does not appear for some $j < i$, we let $j = 0$.

Proof. Notice that U_j is a strict prefix of U_i, and thanks to Lemma 12.9, all the unmatched writes on data-structure d are matched in U_{j+1} since $U_j \xrightarrow{d} U_{j+1}$. Since U_i contains U_{j+1}, U_i does not have any unmatched write on d from the prefix U_j. Hence all the unmatched writes on data-structure d in U_i must be in $U_i \setminus U_j$. □

We can observe that $i - j \leq \mathfrak{d}$, where i and j are as in Lemma 12.10. This maximum value ($\mathfrak{d}$) is reached for those choices of i and d such that $U_i \xrightarrow{d} U_{i+1}$ appears in the sequence. Thus, *all* the unmatched writes on *all* data-structures in the prefix U_j will definitely be matched in U_i where $i = \min(j + \mathfrak{d}, n)$.

Consider any linear extension of the sequence of Lemma 12.10. From the above observation and Lemma 12.9 it follows that the source and the target of any $\rhd$ edge is separated by at most $(\mathfrak{d} + 1)k$ in this linear extension. This gives the witnessing sequence and proves Lemma 12.8. □

Having proved Lemma 12.8, we now show how to bound the split-width.

$\mathsf{sBS}(m)$ We say that an MSCN $\mathcal{M}$ is in **strong $\mathsf{BS}(m)$**, abbreviated $\mathsf{sBS}(m)$, if $\mathcal{M}$ admits a context decomposition $\mathcal{M}\!\!\!/$ with $G_{\mathcal{M}\!\!\!/}$ satisfying the following conditions.

SRW Each context accesses at most one data-structure.

DASL $G_{\mathcal{M}\!\!\!/}$ is direcred acyclic with simple stack self-loops.

S-BS There exists a topological sorting $x_1, x_2, \ldots$ of the contexts such that if $(x_i, x_j) \in E = \bigcup_{d \in \mathbf{DS}} E_d$, then $i \leq j < i + m$.

The first two conditions can together be stated in different words as follows: A context can access either at most one stack or at most one queue in one direction (i.e., either writing mode or reading mode).

Notice that the above conditions subsume the requirements of $\mathsf{BS}(m)$.

An $\mathsf{sBS}(m)$ context-decomposition $\mathcal{M}\!\!\!/$ may have unbounded number of contexts. For our proof, we often focus on the first few contexts of $\mathcal{M}\!\!\!/$, and would like to treat the remaining as a single component. For this, we define an n-$\mathsf{mask}(\mathcal{M}\!\!\!/)$.

Let $\mathcal{M}\!\!\!/$ be a split-MSCN satisfying $\mathsf{sBS}(m)$. Let $x_1, \ldots$ be the topological sorting of the contexts as guaranteed by Condition 12.2. The n-mask of $\mathcal{M}\!\!\!/$

n-$\mathsf{mask}(\mathcal{M}\!\!\!/)$ denoted *n-mask($\mathcal{M}\!\!\!/$)* is the split-MSCN $\mathcal{N}\!\!\!/ \in \mathsf{merge}(\ldots \mathsf{merge}(\mathcal{M}\!\!\!/) \ldots)$ such that all the elastic edges of $\mathcal{M}\!\!\!/$ which do not originate from the first n contexts are transformed into rigid edges in $\mathcal{N}\!\!\!/$. That is, if $\mathcal{M}\!\!\!/ = (\mathcal{M}, \xrightarrow{e})$ then n-$\mathsf{mask}(\mathcal{M}\!\!\!/) =$

$(\mathcal{M}, \xrightarrow{e}')$ where $\xrightarrow{e}' \subseteq \xrightarrow{e}$ is such that, if $(e_1, e_2) \in \xrightarrow{e}'$ then $e_1 \in x_i$ for some $i \leq n$.

Thus the components of n-$\mathsf{mask}(\mathcal{M})$ are $x_1, \ldots, x_n, y_1, \ldots y_{\mathfrak{p}}$, where y_p denotes the single component formed by the merge of all the remaining components in process p.

The elasticity of n-$\mathsf{mask}(\mathcal{M})$ is at most n.

Lemma 12.11. *Let $\mathcal{M}$ be an MSCN in $\mathsf{sBS}(m)$. Then the split-width of $\mathcal{M}$ is at most $m+2$.*

Proof. We will inductively give split-expressions for split-MSCNs showing the bound. The main step in the induction is to identify smaller split-MSCNs allowing the induction to proceed.

In order to give the split-expression $s_{\mathcal{M}}$ for $\mathcal{M}$, consider the split-MSCN $\mathcal{M}$ witnessing the membership of $\mathcal{M}$ in $\mathsf{sBS}(m)$. Also consider the topological sorting $x_1, x_2, \ldots$ of the contexts. It is too expensive to get a split-expression for $\mathcal{M}$. The elasticity of $\mathcal{M}$ is unbounded as $\mathcal{M}$ may have an unbounded number of contexts. Hence, we will rather get a split-expression for m-$\mathsf{mask}(\mathcal{M})$.

Let $\mathcal{M}'$ be the m-mask of $\mathcal{M}$. Clearly $\mathcal{M} \in \mathsf{merge}(\ldots \mathsf{merge}(\mathcal{M}') \ldots)$. Note that the elasticity of m-$\mathsf{mask}(\mathcal{M})$ is at most m. Hence it is sufficient to find a split-expression for m-$\mathsf{mask}(\mathcal{M})$ with width at most $m+2$.

Claim 12.12. *Let $\mathcal{M}'$ be m-$\mathsf{mask}(\mathcal{M})$ for some $\mathsf{sBS}(m)$ context-decomposition $\mathcal{M}$. Then the split-width of $\mathcal{M}'$ is at most $m+2$.*

Proof. The proof is by induction on the size (number of events) of $\mathcal{M}'$. We will inductively give a split-expression $s_{\mathcal{M}'}$ for $\mathcal{M}'$ witnessing the width of $m+2$.

A base case is when $\mathcal{M}'$ has trivial components. In this case $s_{\mathcal{M}'}$ is the shuffle of several basic split-MSCNs. Since the elasticity of $\mathcal{M}'$ is at most m, the split-width of $s_{\mathcal{M}'}$ is at most m.

Another base case is when $\mathcal{M}'$ is a nested-word w. In this case $s_{\mathcal{M}'} = s_w$ as explained in Example 3.23. Recall that the split-width if s_w is at most 2.

We now consider the inductive cases.

Case 1 If $\mathcal{M}'$ has a brink e which does not take part in $\rhd$ relation: Let $\mathcal{M}_1$ be $\mathcal{M}$ without e. Further let $\mathcal{M}'_1$ be the m-$\mathsf{mask}(\mathcal{M}_1)$.[2] We let

$$s_{\mathcal{M}'} = \begin{cases} s_{\mathcal{M}'_1} \sqcup\!\sqcup\, s_e & \text{if } e \text{ forms a component of } \mathcal{M}' \text{ by itself.} \\ \mathsf{merge}(s_{\mathcal{M}'_1} \sqcup\!\sqcup\, s_e) & \text{otherwise.} \end{cases}.$$

The elasticity of the shuffle node is at most $m+1$. Since $\mathcal{M}'_1$ is the m-mask of some $\mathsf{sBS}(m)$ context decomposition $\mathcal{M}_1$, the split-width of $s_{\mathcal{M}'}$ is at most $m+2$ by induction.

[2]Note that $\mathcal{M}'_1$ need not be same as $\mathcal{M}'$ without e. In case e alone forms a component, taking m-$\mathsf{mask}(\mathcal{M}_1)$ would change some y_p by extracting out x_{m+1} of $\mathcal{M}$.

Case 2 If $\mathcal{M}'$ has two brinks e_1 and e_2 linked by a $\rhd$ edge (that is, $e_1 \rhd e_2$) (cf. Case 2 - page 139 and page 146): Let $\mathcal{M}_1$ be $\mathcal{M}$ without e_1 and e_2. Further let $\mathcal{M}'_1$ be the m-$\mathsf{mask}(\mathcal{M}_1)$.[3] We let

$$s_{\mathcal{M}'} = \begin{cases} s_{\mathcal{M}'_1} \sqcup s_{e_1 \rhd e_2} & \text{if both } e_1 \text{ and } e_2 \text{ form} \\ & \text{a component of } \mathcal{M}' \text{ by itself,} \\ \mathsf{merge}(s_{\mathcal{M}'_1} \sqcup s_{e_1 \rhd e_2}) & \text{if either } e_1 \text{ or } e_2 \text{ forms} \\ & \text{a component of } \mathcal{M}' \text{ by itself,} \\ \mathsf{merge}(\mathsf{merge}(s_{\mathcal{M}'_1} \sqcup s_{e_1 \rhd e_2})) & \text{otherwise.} \end{cases}$$

The elasticity of the shuffle node is at most $m+2$. Since $\mathcal{M}'_1$ is the m-mask of some $\mathsf{sBS}(m)$ context decomposition $\mathcal{M}_1$, the split-width of $s_{\mathcal{M}'}$ is at most $m+2$ by induction.

Case 3 If $\mathcal{M}'$ has a brink e_1 in the first n components which is $\rhd$ linked to an event e_2 in the same component: Necessarily the component of e_1, call it x, is accessing a stack d, as otherwise it forms a forbidden queue-loop. The factor of x between e_1 and e_2 including both is a nested-word, call it w[4]. Let $\mathcal{M}'_1$ be $\mathcal{M}'$ without w. We let $s'_{\mathcal{M}} = \mathsf{merge}(s_{\mathcal{M}'_1} \sqcup s_w)$.

The above three cases exhaust all the cases in fact. We argue this below:

First we argue that, if the first two cases are not applicable, then the first component must be accessing a stack.

Suppose the first component is accessing a queue d. It is necessary writing to d as it is the first component. Let e_1 be the first event of the first component. e_1 is necessarily a write event, as otherwise this is in Case 1. Let $e_1 \rhd e_2$ and let x be the component of e_2. x is a component which reads from d. Necessarily e_2 must be the first read event of x in order to comply with the FIFO policy on queue d. In fact e_2 must also be the first event of x, as otherwise it would have been in Case 1. If e_2 is the first event then e_2 is also a brink which should be handled in Case 2. This contradicts the assumption that Case 1 and Case 2 are not applicable. Thus the first context must not be accessing a queue.

Now we argue that if the first component is accessing a stack and if none of the cases are applicable, we reach a contradiction.

Suppose the first component is accessing a stack d. Let e_1 be its first event of the first component and let $e_1 \rhd e_2$. The event e_2 is necessarily in another component x, as otherwise Case 3 is applicable. If e_2 is a brink of x, case 2 is applicable, hence e_2 is not a brink. Let e_3 be the first event of x, and let S be the set of components smaller than (in the topological sorting) x accessing d. Let B denote the set of

[3]Note that $\mathcal{M}'_1$ need not be same as $\mathcal{M}'$ without e_1 and e_2. If any of these events form a single component, taking m-$\mathsf{mask}(\mathcal{M}_1)$ would change some y_p by extracting out x_{m+1} of $\mathcal{M}$.

[4]w is a strict factor of x, as otherwise e_2 is also a brink and hence would have been handled in Case 2.

brinks of S and e_3. All the event in B must have a $\rhd$ partner as otherwise, it is case 1. None of the events in B has their partner in their own component, as otherwise it is Case 3. All the $\rhd$ partners of B must be in S, as otherwise it will violate the LIFO policy on d with (e_1, e_2). Notice, in particular, that e_3 is a read event and its partner is also in S (if e_3 was a write event, its partner must be in the same component x before e_2 which is a Case 3 instance). Consider an innermost $\rhd$ edge between some event e in B and its partner e'. Both e and e' must be brinks from B. If not, let e'' be the brink of the component of e' which is in between e and e'. Since $e'' \in B$, its $\rhd$ edge must be the innermost contradicting the assumption the $e \rhd e'$ is an innermost $\rhd$ edge. Hence e and e' must be brinks contradicting the assumption that Case 2 is not applicable.

Thus the proof of Claim 12.12 is complete. □

From Claim 12.12, the split-MSCN $\cancel{\mathcal{M}}$ has a split-expression of width at most $m + 2$. This proves Lemma 12.11. □

From Lemma 12.8 we get the following:

Proposition 12.13. *If* $\mathcal{M} \in \mathsf{BS}(k)$ *then* $\mathcal{M} \in \mathsf{sBS}((\mathfrak{d} + 1)k)$.

From Proposition 12.13 and Lemma 12.11 together prove the the bound of $(\mathfrak{d} + 1)k + 2$ on the split-width of $\mathsf{BS}(k)$. □

Remark 12.14. The above bound can be improved to $(\mathfrak{d}+1)k+1$ by considering $(m - 1)$-masks instead of m-masks.

Remark 12.15. The split-expression we obtain for MSCNs in $\mathsf{BS}(k)$ are not word-like. However, if the architecture does not have stacks (like in communicating finite state machines), the split-expressions are word-like.

Remark 12.16. Notice that whenever the word-like property is violated at a shuffle node, one of its children is s_w for some nested-word w. The split-width of s_w is at most 2 (cf. Example 3.23). Thus the split-terms we obtain for $\mathsf{BS}(k)$ are *almost-word-like*. A split-term is almost-word-like if at every shuffle node, the split-width of one of its children is bounded by a constant m. `almost-word-like`

12.3 Controller

12.3.1 Controller description

For $\mathsf{BS}(k)$ we propose an infinite state deterministic CPDS as a controller. The local states of the controller contain global context-vectors (best knowledge of a local process about the global context). These are passed around by data-structure accesses, which allows the controllers 1) to update the current context vector (or its best knowledge about the global context) and 2) to compare the context-vectors so that the bound on scope can be checked.

We will see that change of the current context can be detected deterministically. A (greedy) strategy is not to change the current context unless necessary.

The contexts are numbered sequentially (cf. the canonical naming of the contexts). The controller remembers the current context number as well as the best information it has about the contexts of the other processes (global knowledge). This is represented as a 'context-vector' ($\overrightarrow{cv}$). A *context-vector* is a tuple of non-negative integers ($(i_p)_{p \in \mathbf{Procs}}$) which gives a context-number for each process.

`context-vector`

A write-event writes the current context-vector to the data-structure so that the controller can access it at the corresponding read for comparisons.

For each process, a change of context is inevitable if there is a change of the data-structure it has been accessing, or in the case of queue data-structures, if the access mode (read / write) changes. Hence the controller keeps this information ($\texttt{curr-DS} \in \mathbf{Stacks} \cup (\mathbf{Queues} \times \{\text{write, read}\})$) in its local state.

Moreover, it needs to check whether the bound of k contexts has been exceeded since the earliest write-context to the current-context. If the current context is accessing a stack, the context-vector of the earliest write is the one available on the top of stack (due to the LIFO policy).

However, if the current context is reading a queue, then the earliest write-context to the current context corresponds to the context of the first write to the current context. This information is not available on later read events. Hence on the first read event of a queue-read-context the controller will store the context-vector at the corresponding write also in its memory. This is denoted ($\overrightarrow{iv}$). If the controller detects that prolonging the current context will add more than k contexts between the current context and $\overrightarrow{iv}$, it will again switch context.

More formally, the local states of the controller for process p consist of three parts: A current-data-structure (`curr-DS`), a current-context-vector ($\overrightarrow{cv}$) and an initial-context-vector ($\overrightarrow{iv}$). Current-data-structure (`curr-DS`) remembers which data-structure is being accessed in the current context, and if the data-structure is a queue, also the direction of access. The initial-context-vector is needed only if the current context is accessing a queue in read-mode. It is maintained as $\perp$ in other cases.

The initial control location of every local controller is the tuple

$$\texttt{curr-DS} = \perp, \overrightarrow{cv} = (0, \ldots, 0), \overrightarrow{iv} = \perp.$$

As we said before, on a read-event the controller needs to access the context-vector at the write-event, in order to ensure bounded scope. For this, it will also "context-stamp" the data-structure entries. On a write to a data-structure, the writer's best knowledge about the contexts ($\overrightarrow{cv}$) is also written into the data-structure. This will allow the controller to update its global knowledge at a read-event. On a read-event,

1. it asserts that the bounded scope is respected by comparing the (updated) current context-vector, and the context-vector of the earliest write-event

to the current context. The latter is either available in the data-structure entry, or in the local state, depending on whether it is a stack, or a queue, that is being accessed.

2. if the above assertion does not hold by prolonging the current context, it checks if it can be held by forcing a change of context. This case is meaningful only in the case of a queue-read context. Note that, if there is a change of context then the earliest-write-context ($\overrightarrow{iv}$) is updated. Hence there could be potentially fewer contexts between the current context and the updated $\overrightarrow{iv}$.
3. If the assertion fails, the run is aborted.
4. If the assertion holds, the current context vector is updated.

We will now describe the transitions formally. For $\overrightarrow{cv} = (i_q)_{q \in \mathbf{Procs}}$, we define $\mathsf{inc}_p(\overrightarrow{cv}) = (i'_q)_{q \in \mathbf{Procs}}$ where $i'_q = \begin{cases} i_q & \text{if } q \neq p \\ 1 + i_p & \text{otherwise.} \end{cases}$ $\mathsf{inc}_p(\overrightarrow{cv})$

The transitions preserve the local state if not accessing a data-structure.

$$((\texttt{curr-DS}, \overrightarrow{cv}, \overrightarrow{iv}), a, (\texttt{curr-DS}, \overrightarrow{cv}, \overrightarrow{iv})) \in \text{Trans}_{p:int} \tag{12.1}$$

The write transitions update the state (both $\texttt{curr-DS}$ and $\overrightarrow{cv}$) if a change of context is detected. Also, the updated context-vector $\overrightarrow{cv}'$ is always written onto the data-structure.

$$((\texttt{curr-DS}, \overrightarrow{cv}, \overrightarrow{iv}), a, (\texttt{curr-DS}', \overrightarrow{cv}', \bot), \overrightarrow{cv}') \in \text{Trans}_{p \to d} \tag{12.2}$$

where

$$\texttt{curr-DS}' = \begin{cases} (d, \text{write}) & \text{if } d \in \mathbf{Queues} \\ d & \text{if } d \in \mathbf{Stacks} \end{cases}$$

and, with $\overrightarrow{cv} = (i_q)_{q \in \mathbf{Procs}}$,

$$\overrightarrow{cv}' = \begin{cases} \overrightarrow{cv} & \text{if } \texttt{curr-DS} = \texttt{curr-DS}' \\ \mathsf{inc}_p(\overrightarrow{cv}) & \text{otherwise.} \end{cases}$$

Notice that the component $\overrightarrow{iv}$ is not used in the above transitions. Also, the context-vectors are not compared. Thus on internal events and write events, the controller simply increments the current context if a change of context is detected (via violation of SRW or DASL). On write events, it writes the current context-vector onto the data-structure.

It is in fact read transitions which ensure that the bounded scope restriction is respected. For easily defining the transitions in this case, we first define two operations on context-vectors.

Let $\overrightarrow{cv}' = (i'_p)_{p \in \mathbf{Procs}}$ and $\overrightarrow{cv}'' = (i''_p)_{p \in \mathbf{Procs}}$. We define $\max(\overrightarrow{cv}', \overrightarrow{cv}'')$ to be the context-vector $\overrightarrow{cv} = (i_p)_{p \in \mathbf{Procs}}$ where $i_p = \max(i'_p, i''_p)$. $\max(\overrightarrow{cv}', \overrightarrow{cv}'')$

$\|\overrightarrow{cv}' - \overrightarrow{cv}''\|$ By $\|\overrightarrow{cv}' - \overrightarrow{cv}''\|$ we denote the sum of the component-wise differences. That is, $\|\overrightarrow{cv}' - \overrightarrow{cv}''\| = \sum_{p \in \mathbf{Procs}} i'_p - i''_p$. We may assume that $\overrightarrow{cv}''$ is component-wise smaller than $\overrightarrow{cv}'$ as we need only such cases.

We first consider the easier case of a transition reading from a stack, and then consider the case of queue.

The stack-read transitions update the state if a change of context is detected. Only those read transitions which respect the scope bounded restriction are permitted.

For $d \in \mathbf{Stacks}$, we have a transition of the following form if $\|\overrightarrow{cv}' - \overrightarrow{cv}''\| < k$:

$$((\texttt{curr-DS}, \overrightarrow{cv}, \overrightarrow{iv}), \overrightarrow{cv}'', a, (d, \overrightarrow{cv}', \bot)) \in \mathrm{Trans}_{p \leftarrow d} \tag{12.3}$$

where,

$$\overrightarrow{cv}' = \begin{cases} \overrightarrow{cv} & \text{if } \texttt{curr-DS} = d \\ \mathsf{inc}_p(\overrightarrow{cv}) & \text{otherwise.} \end{cases}$$

Note that $\overrightarrow{cv}'' \leq \overrightarrow{cv}$ component-wise.

The queue-read transitions also update the state if a change of context is detected. It will also change the context if prolonging the current context violates the bounded scope restriction. For this, we use the initial-context-vector $\overrightarrow{iv}$. Notice that $\overrightarrow{iv}$ has a value other than $\bot$ only if $\texttt{curr-DS}$ is (d, read).

For $d \in \mathbf{Queues}$, we have a transition of the following form if $\|\overrightarrow{cv}' - \overrightarrow{iv}'\| < k$:

$$((\texttt{curr-DS}, \overrightarrow{cv}, \overrightarrow{iv}), \overrightarrow{cv}'', a, ((d, \mathrm{read}), \overrightarrow{cv}', \overrightarrow{iv}')) \in \mathrm{Trans}_{p \leftarrow d} \tag{12.4}$$

where,

$$\overrightarrow{cv}' = \begin{cases} \max(\overrightarrow{cv}, \overrightarrow{cv}'') & \text{if } \texttt{curr-DS} = (d, \mathrm{read}) \text{ and } \overrightarrow{cv}' - \overrightarrow{iv} < k \\ \mathsf{inc}_p(\max(\overrightarrow{cv}, \overrightarrow{cv}'')) & \text{otherwise} \end{cases}$$

and $\overrightarrow{iv}' = \begin{cases} \overrightarrow{iv} & \text{if } \texttt{curr-DS} = (d, \mathrm{read}) \text{ and } \overrightarrow{cv}' - \overrightarrow{iv} < k \\ \overrightarrow{cv}'' & \text{otherwise.} \end{cases}$

Notice that $\overrightarrow{iv}'$ is the context-vector at the earliest write-event to the context (p, i'_p). Also $\overrightarrow{cv}'$ covers all the context in the past of a read event. In particular, $\overrightarrow{iv}' \leq \overrightarrow{cv}'$ component-wise. Hence $\|\overrightarrow{cv}' - \overrightarrow{iv}'\| < k$ ensures BS.

The controller permits only safe transitions (those respecting the bounded scope restriction). Hence, the global acceptance set may allow any (all) combinations of states.

The set of control locations of the controller is $(\mathbf{Queues} \times \{\mathrm{read}, \mathrm{write}\} \cup \mathbf{Stacks} \cup \{\bot\}) \times \mathbb{N}^{\mathbf{Procs}} \times (\mathbb{N}^{\mathbf{Procs}} \cup \bot)$.

The controller can be 'completed' by adding a sink state called $\texttt{ABORT}$. The above controller is denoted $\mathcal{S}_{\mathsf{BS}(k)}$.

12.3.2 Soundness and Completeness

The deterministic infinite state controller $\mathcal{S}_{\mathsf{BS}(k)}$ based on context-stamping is sound and complete.

Proposition 12.17. *$\mathcal{S}_{\mathsf{BS}(k)}$ is sound and complete for* $\mathsf{BS}(k)$.

Proof. The soundness and completeness are proved respectively in Lemma 12.18 and Lemma 12.19. □

Lemma 12.18. $\mathscr{L}(\mathcal{S}_{\mathsf{BS}(k)}) \subseteq \mathsf{BS}(k)$.

Proof. If an MSCN $\mathcal{M}$ has an accepting run in $\mathcal{S}_{\mathsf{BS}(k)}$, then a context-decomposition $\mathcal{M}$ is suggested by the c-loc mapping of the run. The context-graph $G_{\mathcal{M}}$ belongs to $\mathcal{G}_{\mathsf{BS}(k)}$ since the transitions of $\mathcal{S}_{\mathsf{BS}(k)}$ ensure the conditions SRW, DASL and BS, as explained below.

SRW: Note that the transitions update $\overrightarrow{cv}$ whenever `curr-DS` is updated. Thus only one data-structure can be accessed in a context.

DASL: We identify a queue context together with its access mode. Hence no queue loops are allowed in a single context. Now DASL follows from SRW and directed acyclicity of MSCNs.

BS: If the condition BS were violated, then there would be $e_1 \rhd e_2$ with e_1 and e_2 in contexts x and y respectively such that the number of contexts in the past of y but not in the past of x are more than k. The transitions ensure that this cannot happen as we explain below.

In the case of a stack data-structure, it compares the context vector (say $\overrightarrow{cv}_1$) at e_1 (which is available at the top of the stack) with the context vector (say $\overrightarrow{cv}_2$) at e_2. Notice that no new event can be added to the past of $\overrightarrow{cv}_1$ (respectively $\overrightarrow{cv}_2$) by events occuring later than e_1 (respectively e_2).

For a queue data-structure, on the other hand, it compares $\overrightarrow{cv}$ with $\overrightarrow{iv}$ on every read event of y. Since $\overrightarrow{iv}$ is the context vector at the earliest write to y, $\overrightarrow{iv} \leq \overrightarrow{cv}_1$ component-wise. Since the comparison is made at every read-event of y, there is a comparison in which $\overrightarrow{cv} \geq \overrightarrow{cv}_2$. Hence the transitions ensure that BS is respected.

Thus $\mathcal{M}$ is a witnessing context-decomposition for the membership in $\mathsf{BS}(k)$. □

Lemma 12.19. $\mathsf{BS}(k) \subseteq \mathscr{L}(\mathcal{S}_{\mathsf{BS}(k)})$.

Proof. If an MSCN $\mathcal{M}$ belongs to $\mathsf{BS}(k)$, it has a witnessing context-decomposition which is also 'greedy'. We will describe below how any abritrary $\mathsf{BS}(k)$ context-decomposition of $\mathcal{M}$ can be transformed into a greedy one by prolonging the contexts as long the condition BS is not violated. Such a greedy $\mathsf{BS}(k)$ context-decomposition of $\mathcal{M}$ dictates an accepting run of the controller $\mathcal{S}_{\mathsf{BS}(k)}$.

Let $\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \delta, \xrightarrow{\mathrm{r}}, \xrightarrow{\mathrm{e}}, \rhd)$ be a context-decomposition of $\mathcal{M}$ with $G_{\mathcal{M}} = (V, (E_d)_{d\in\mathbf{DS}}) \in \mathcal{G}_{\mathsf{BS}(k)}$ its extended context-graph.

For a context $x \in V$, we denote its 'type' by $\texttt{curr-DS}(x)$. We have $\texttt{curr-DS}(x) \in \textbf{Stacks} \cup (\textbf{Queues} \times \{\text{read}, \text{write}) \cup \{\bot\}$. If a context x is not accessing any data-structure, the $\texttt{curr-DS}(x) = \bot$.

For an event $e \in \mathcal{E}$, we denote by $\mathsf{context}_{G_{\mathcal{M}}}(e)$ the component $x \in V$ such that $e \in x$. When the context-decomposition is clear, we may simply write $\mathsf{context}(e)$. We lift this naturally to sets of events as well: For $\mathcal{E}' \subseteq \mathcal{E}$, $\mathsf{context}_{G_{\mathcal{M}}}(\mathcal{E}') = \cup_{e \in \mathcal{E}'} \mathsf{context}_{G_{\mathcal{M}}}(e)$.

We denote the contexts of $G_{\mathcal{M}}$ in the past of an event e by $\mathsf{past}_{G_{\mathcal{M}}}(e)$ and the contexts in the past of $\mathsf{context}(e)$ by $\mathsf{PAST}_{G_{\mathcal{M}}}(e)$:

- $\mathsf{past}_{G_{\mathcal{M}}}(e) = \{x \in V \mid \exists e' \leq e \text{ such that } x = \mathsf{context}(e')\}$
- $\mathsf{PAST}_{G_{\mathcal{M}}}(e) = \{x \in V \mid x \preccurlyeq \mathsf{context}(e)\} = \{x \in V \mid \exists e'' \in \mathsf{context}(e) \text{ and } e' \leq e'' \text{ such that } x = \mathsf{context}(e')\}$

Also, we denote the downward closure in $G_{\mathcal{M}}$ of a context x by $\downarrow_{G_{\mathcal{M}}} x = \{y \mid y \preccurlyeq x\}$. The downward closure of an event $\downarrow e = \{f \in \mathcal{E} \mid f \leq e\}$.

Observe that $\mathsf{past}_{G_{\mathcal{M}}}(e) \subseteq \mathsf{PAST}_{G_{\mathcal{M}}}(e) = \downarrow_{G_{\mathcal{M}}} \mathsf{context}(e)$.

Recall that $G_{\mathcal{M}} \in \mathcal{G}_{\mathsf{BS}(k)}$ if and only if for every $\rhd$ edge $f_1 \rhd f_2$,

$$|\mathsf{PAST}_{G_{\mathcal{M}}}(f_2) \setminus \mathsf{PAST}_{G_{\mathcal{M}}}(f_1)| < k.$$

Now we will explain how to obtain a greedy context-decomposition $\mathcal{M}'$ with $G_{\mathcal{M}'} \in \mathsf{BS}(k)$ from an arbitrary context-decomposition $\mathcal{M}$ with $G_{\mathcal{M}} \in \mathsf{BS}(k)$. This is done in two phases.

Two contexts x and y are consecutive if with the canonical naming of the contexts, $x = (p, i)$ and $y = (p, i+1)$ for some process $p \in \textbf{Procs}$. In the first phase, consecutive pairs of contexts x and y are merged into a single context if

1. If $\texttt{curr-DS}(x) = \bot$ and y is arbitrary.
2. If $\texttt{curr-DS}(y) = \bot$ and x is arbitrary.
3. Both x and y are accessing the same stack. That is $\texttt{curr-DS}(x) = \texttt{curr-DS}(y) = d$ for some $d \in \textbf{Stacks}$.
4. Both x and y are writing to the same queue. That is $\texttt{curr-DS}(x) = \texttt{curr-DS}(y) = (d, \text{write})$ for some $d \in \textbf{Queues}$.

Notice that the first phase does not increase the number of contexts between $\mathsf{context}(f_1)$ and $\mathsf{context}(f_2)$ for any $f_1 \rhd f_2$. Hence the resulting context-decomposition after the first phase, call it $\mathcal{M}'$, again witnesses $\mathsf{BS}(k)$:

$$|\mathsf{PAST}_{G_{\mathcal{M}'}}(f_2) \setminus \mathsf{PAST}_{G_{\mathcal{M}'}}(f_1)| \leq |\mathsf{PAST}_{G_{\mathcal{M}}}(f_2) \setminus \mathsf{PAST}_{G_{\mathcal{M}}}(f_1)| < k.$$

That is $G_{\mathcal{M}'} \in \mathcal{G}_{\mathsf{BS}(k)}$.

In the second phase we deal with adjascent queue-reading contexts. We may assume that compatible queue-write contexts and stack contexts have already been merge (i.e., first phase is not applicable). Let x and y be two consecutive contexts reading from a queue $d \in \mathbf{Queues}$. We show that if the first event e of y is feasible for x, then detaching e from y and attaching it as the last event of x still preserves bounded scope. Thus by induction, we may conclude that we can obtain a greedy decomposition for $\mathsf{BS}(k)$ from an arbitrary one.

Let x and y be two adjascent contexts (say $x = (p, i)$ and $y = (p, i+1)$ with $\texttt{curr-DS}(x) = \texttt{curr-DS}(y) = (d, \text{read})$ and let e be the first event of y. Let e_2 be the first read on x and let e_1 be the matching write (that is $e_1 \rhd^d e_2$, and $e_2 \in x$ is the first read-event of x.[5])

We say the event e is *feasible* for x if $|(\mathsf{past}_{G_{\mathcal{M}}}(e) \setminus \{y\}) \setminus \mathsf{PAST}_{G_{\mathcal{M}}}(e_1)| < k$. `feasible`
If there is an event e which is feasible for x, then the context-decomposition is not greedy. Note that the controller $\mathcal{S}_{\mathsf{BS}(k)}$ does not change the context (and context-vector) at a feasible event, thus permitting only greedy context-decomposition.

Let $\mathcal{M}'$ be the context-decomposition obtained from $\mathcal{M}$ by detaching a feasible event e from context y and attaching it to x. We will now argue that, $\mathcal{M}'$ still verifies $\mathsf{BS}(k)$, that is, $G_{\mathcal{M}'} \in \mathcal{G}_{\mathsf{BS}(k)}$. For this, it suffices to show that for all edges $f_1 \rhd f_2$,

$$|\mathsf{PAST}_{G_{\mathcal{M}'}}(f_2) \setminus \mathsf{PAST}_{G_{\mathcal{M}'}}(f_1)| < k.$$

In the following two claims we relate the downward closures in $G_{\mathcal{M}}$ and $G_{\mathcal{M}'}$ of a context.

Claim 12.20. $\downarrow_{G_{\mathcal{M}'}} x = \mathsf{past}_{G_{\mathcal{M}}}(e) \setminus \{y\}$.

Proof. Since the maximal event of x in $\mathcal{M}'$ is e. $\square$

Claim 12.21. *If $y \neq \{e\}$ in $\mathcal{M}$, then for all contexts $z \neq x$, $\downarrow_{G_{\mathcal{M}}} z = \downarrow_{G_{\mathcal{M}'}} z$.*

Proof. We make the following observations:

OBS1 For all events $g \neq e$, $\mathsf{context}_{G_{\mathcal{M}}}(g) = \mathsf{context}_{G_{\mathcal{M}'}}(g)$.

OBS2 For the event e, $\mathsf{context}_{G_{\mathcal{M}}}(e) = y$ and $\mathsf{context}_{G_{\mathcal{M}'}}(e) = x$.

From the above observations, we get the following observation as well.

OBS3 $\mathsf{context}_{G_{\mathcal{M}}}(\downarrow f \setminus e) = \mathsf{context}_{G_{\mathcal{M}'}}(\downarrow f \setminus e)$.

OBS4 For all split-MSCNs $\mathcal{M}''$ with $G_{\mathcal{M}''} \in \mathcal{G}_{\mathsf{BS}(k)}$ and all contexts v, letting f be the maximal event of v,

$$\downarrow_{G_{\mathcal{M}''}} v = \mathsf{past}_{G_{\mathcal{M}''}}(f) = \mathsf{context}_{G_{\mathcal{M}''}}(\downarrow f)$$

This is true in particular for $\mathcal{M}$ and $\mathcal{M}'$ as well.

[5]If 'greedification' chooses the left-most non-greedy event each time, then the event e_2 is indeed the first event of x.

OBS5 If $e < g$ for some event g, then since e is a read event, there must be an event e' successor of e such that $e \rightarrow e' \leq g$. Since $y \neq \{e\}$ in $\mathcal{M}$, $\mathsf{context}_{G_{\mathcal{M}'}}(e') = y$.

Having made the above observations, we are now ready to prove the claim. Let f be the maximal event of context z. Note that it is the same in both $\mathcal{M}$ and $\mathcal{M}'$, as $z \neq x$. We have two cases to consider.

1. If $e \not\leq f$:

   Thanks to OBS3 and OBS4, the claim follows.

2. If $e \leq f$: Since $z \neq x$, $e \lneq f$. From OBS5, both x and y are in $\mathsf{context}_{G_{\mathcal{M}}}(\downarrow f)$ as well as $\mathsf{context}_{G_{\mathcal{M}'}}(\downarrow f)$. Therefore, $\downarrow_{G_{\mathcal{M}}} z = \downarrow_{G_{\mathcal{M}'}} z$ if $z \neq x$ and $y \neq \{e\}$ in $\mathcal{M}$. □

Claim 12.22. *Suppose $y = \{e\}$ in $\mathcal{M}$.*

1. *If $e \leq g$, then $\mathsf{context}_{G_{\mathcal{M}'}}(\downarrow g) = \mathsf{context}_{G_{\mathcal{M}}}(\downarrow e) \setminus \{y\}$.*
2. *If $e \not\leq g$, then $\mathsf{context}_{G_{\mathcal{M}'}}(\downarrow g) = \mathsf{context}_{G_{\mathcal{M}}}(\downarrow e)$.*

Now consider $f_1 \rhd f_2$. We have two cases depending on whether the context of f_2 in $G_{\mathcal{M}'}$ is x or not. Notice that $x \neq \mathsf{context}(f_1)$ as x is a queue-read context and f_1 is a write event.

- If x is not the context of f_2 in $G_{\mathcal{M}'}$:

  1. If $y \neq \{e\}$ in $\mathcal{M}$.
     From Claim 12.21 we know that $\mathsf{PAST}_{G_{\mathcal{M}}}(f_2) = \mathsf{PAST}_{G_{\mathcal{M}'}}(f_2)$ and $\mathsf{PAST}_{G_{\mathcal{M}}}(f_1) = \mathsf{PAST}_{G_{\mathcal{M}'}}(f_1)$.
     Hence $|\mathsf{PAST}_{G_{\mathcal{M}}}(f_2) \setminus \mathsf{PAST}_{G_{\mathcal{M}}}(f_1)| = |\mathsf{PAST}_{G_{\mathcal{M}'}}(f_2) \setminus \mathsf{PAST}_{G_{\mathcal{M}'}}(f_1)|$.
     Since $G_{\mathcal{M}} \in \mathcal{G}_{\mathsf{BS}(k)}$, the above cardinality is bounded by k.
  2. $y = \{e\}$ in $\mathcal{M}$. If $e \not\leq g$, then again $|\mathsf{PAST}_{G_{\mathcal{M}'}}(f_2) \setminus \mathsf{PAST}_{G_{\mathcal{M}'}}(f_1)| = |\mathsf{PAST}_{G_{\mathcal{M}}}(f_2) \setminus \mathsf{PAST}_{G_{\mathcal{M}}}(f_1)| < k$.
     If $e \leq g$, then $|\mathsf{PAST}_{G_{\mathcal{M}'}}(f_2) \setminus \mathsf{PAST}_{G_{\mathcal{M}'}}(f_1)| \leq |\mathsf{PAST}_{G_{\mathcal{M}}}(f_2) \setminus \mathsf{PAST}_{G_{\mathcal{M}}}(f_1)| < k$.

- If x is the context of f_2 in $G_{\mathcal{M}'}$:

  We have $|\mathsf{PAST}_{G_{\mathcal{M}'}}(f_2) \setminus \mathsf{PAST}_{G_{\mathcal{M}'}}(f_1)| \leq |\downarrow_{G_{\mathcal{M}'}} x \setminus \mathsf{PAST}_{G_{\mathcal{M}'}}(e_1)|$ due to the FIFO policy on the queue d: Note that e_1 and f_1 write to the same queue, and e_2 and f_2 are the matching read events with $e_2 \leq f_2$.

  Since e is the last event of context x in $\mathcal{M}'$, we get,

  $$\begin{aligned}\downarrow_{G_{\mathcal{M}'}} x \setminus \mathsf{PAST}_{G_{\mathcal{M}'}}(e_1) &= \mathsf{past}_{G_{\mathcal{M}'}}(e) \setminus \mathsf{PAST}_{G_{\mathcal{M}'}}(e_1) \\ &= (\mathsf{past}_{G_{\mathcal{M}}}(e) \setminus \{y\}) \setminus \mathsf{PAST}_{G_{\mathcal{M}'}}(e_1).\end{aligned}$$

  Since e is feasible for x in $\mathcal{M}$, the cardinality of the above set is bounded by k: $|(\mathsf{past}_{G_{\mathcal{M}}}(e) \setminus \{y\}) \setminus \mathsf{PAST}_{G_{\mathcal{M}'}}(e_1)| < k$.

  Thus $|\mathsf{PAST}_{G_{\mathcal{M}'}}(f_2) \setminus \mathsf{PAST}_{G_{\mathcal{M}'}}(f_1)| < k$.

Thus for every $f_1 \rhd f_2$, $|\mathsf{PAST}_{G_{\mathcal{M}'}}(f_2) \setminus \mathsf{PAST}_{G_{\mathcal{M}'}}(f_1)| < k$. Hence $G_{\mathcal{M}'} \in \mathcal{G}_{\mathsf{BS}(k)}$. Now, we repeat the above procedure until the context-decomposition is greedy. This proves completeness of $\mathcal{S}_{\mathsf{BS}(k)}$. □

12.3.3 Discussions

This section leaves out a very interesting problem for future work.

Open Question 12.23. *Does there exist a finite state controller (as a CPDS) for the class* $\mathsf{BS}(k)$*?*

Remark 12.24. In the case of multi-pushdown systems, a finite state controller indeed exists[6]. We will describe it informally.

We may observe that, if there is a violation of BS by an MNW (multiply nested word, which is an MSCN over the architecture of multi-pushdown systems), then there is always a bottom-most stack entry which has stayed for more than k contexts. The controller thus aims to verify that all bottom most entries stay for at most k contexts, and this will ensure bounded scope.

The state of a controller is a tuple $\mathbf{Stacks} \times \{0, 1, \ldots, k\}^{\mathbf{Stacks}}$. A state $(s, (i_d)_{d \in \mathbf{Stacks}})$ indicates that stack which is active in the current context is s, and the bottom-most entry on stack d has seen i_d contexts so far. The bottom-most entry on a stack will always be tagged # and all other entries are tagged $\star$. These are the only two values which will be written on the stacks by the controller.

Observe that the change of context can be decided deterministically in the case of $\mathsf{BS}(k)$. Thus, the controller will deterministically change the context when needed, and increment the values of all (but one, sometimes) i_d. If at any point this incrementation step would exceed the bound k, the controller moves into an `ABORT` state. When it pops # from a stack d, the value of i_d is reset to 0. If there is a push on stack d while i_d is 0, then it pushes # and changes i_d to 1 in the next state. All other pushes are $\star$.

The finite state deterministic controller for multi-pushdown systems is sound and complete. It aborts a run as soon as a violation of scope bounded is detected.

Note that this controller cannot be extended as such to generic architectures as it relies crucially on the fact that there is only one process (which imposes a total order on the contexts), and that there are only stack data-structures.

12.4 Decision Procedures and Discussions

The verification problems become decidable in the case of $\mathsf{BS}(k)$ also, as this class is MSO definable and has bounded split-width. Let $k' = (\mathfrak{d} + 1)k + 2$, denote the bound on split-width for $\mathsf{BS}(k)$.

We cannot use the tree-automaton $\mathcal{A}^{k'}(\mathcal{S}_{\mathsf{BS}(k)})$ for obtaining the complexity bounds, as our controller is not a finite-state CPDS. Instead we will use the

[6] I thank Salvatore La Torre for a fruitful discussion on controllers for multi-pushdown systems.

translation of the MSO formula given in page 150. Call it $\varphi_{\mathsf{BS}(k)}$. The tree-automaton $\mathcal{A}^{k'}(\varphi_{\mathsf{BS}(k)})$ (cf. page 74) recognises the encodings of $\mathsf{BS}(k)$ among valid k'-DSTs.

Claim 12.25. *The size of $\mathcal{A}^{k'}(\varphi_{\mathsf{BS}(k)})$ is doubly exponential in k.*

Proof. Consider the formula $\varphi_{\mathsf{BS}(k)}$ given in Section 12.2. For every atomic binary relation appearing in the formula, we will give a linear sized 2-way walking tree-automaton over k'-DSTs. Those atomic binary relations other than $x < y$ are discussed in Section 4.3.1. We now argue that the relation $x < y$ can also be recognised by a linear sized 2-way walking tree-automaton over k'-DSTs.

For $<$ along a process, the automaton only needs to verify that both x and y have the same pid value, and on their least common ancestor, the component corresponding to x is before the component corresponding to y. Thus, it remembers the component number in its state. As a walking automaton, it would move up tracking the component of x, and then non-deterministically jump to a later component in the same node, and move down tracking this component (it resolves the choice on moving down a merge node in a non-deterministic fashion) until it reaches a leaf labelled y.

For $<$ as partial order, we can still have a walking automaton which alternates between $\rhd$ relation and $<$ along a process at most $\mathfrak{p}$ times. (Actually the number of times it alternates between these two or whether it repeats a same process several times does not matter. We can also take the transitive closure of $\rightarrow + \rhd$ directly.) Thus we have a 2-way walking automaton for $<$ whose number of states is linear in $k\mathfrak{p}$.

Thus for the formula from 2nd line onwards, we have a 2-way alternating automaton whose size is polynomial in k. This automaton accepts trees over a slightly extended alphabet (with x_i, x'_i, y_i, y'_i etc. marked on the leaves). For obtaining the automaton for the existential quantification, we need a projection to the k'-DST-Labels. For this, we first need to convert the polynomial sized 2-way alternating automaton to a non-deterministic tree-automaton. This causes an exponential blow up, giving us a non-deterministic tree-automaton whose size is $2^{\mathrm{poly}(k)}$.

Finally, in order to incorporate the top-most negation, we require a complementation, which adds another exponentiation. Thus we have a doubly exponential sized tree-automaton for $\mathsf{BS}(k)$. Note that, for the complement of $\mathsf{BS}(k)$ within $\mathbb{MSCN}_{k'}$, we have a polynomial sized (in k) alternating 2-way automaton, and an exponential sized (in k) non-deterministic tree-automaton over the k'-DSTs. □

Corollary 12.26. *Consider the set of valid (k')-DST encodings of MSCNs with split-width at most k' which are NOT in $\mathsf{BS}(k)$. There exists a tree-automaton $\mathcal{A}_{\neg\mathsf{BS}(k)}$ over valid k'-DSTs which recognises the above set. The size of $\mathcal{A}_{\mathsf{BS}(k)}$ is exponential in k. The above set can also be recognised by a 2-way alternating tree-automaton whose size is polynomial in k.*

Complexity Upper Bounds The split-width k' of $\mathsf{BS}(k)$ is indeed bounded by some m which is exponential in k: $k' = (\mathfrak{d}+1)k+2 \leq m \in 2^{\text{poly}(k)}$. There is a tree-automaton of size at most $2^{\text{poly}(m)}$ over valid m-DSTs recognising encodings of $\mathsf{BS}(k)$. Hence we get the complexity upper bounds stated in Table 4.5 for the verification problems.

Note that $\mathsf{BS}(k)$ admits word-like split-terms if the architecture does not have stacks. Hence for stack-free architectures, the complexity upper bounds for the verification problems are as stated in Table 4.6.

For multi-pushdown systems we have an exponential sized controller. Thus in this case there is an exponential sized tree-automaton over valid k'-DSTs recognising $\mathsf{BS}(k)$. Moreover k' is polynomial in k. Thus, the verification problems for scope bounded multi-pushdown systems are decidable with complexity upper bounds as stated in Table 4.3.

In fact these upper bounds (Table 4.3) would carry over to arbitrary architectures if we can find an exponential sized controller for $\mathsf{BS}(k)$.

Further (Space) Optimisations To optimise futher, recall that we have an almost-word-like split-term for $\mathsf{BS}(k)$ (Remark 12.16). We will now argue that we can match the space complexity as for word-like by a polynomial time pre-computation.

The idea is, once we obtain a tree-automaton $\mathcal{A}_1$ over almost-word-like k'-DSTs, to compute the reachable set of states of this tree automaton by terms corresponding to nested-words. Let us call this set $R(\mathcal{A}_1)$. This is a pre-computation phase which takes time polynomial in the size of $\mathcal{A}_1$.

Then we obtain another tree-automaton $\mathcal{A}_2$ which is supposed to accept the 'pruned' versions of the almost-word-like k'-DSTs, where terms corresponding to nested-words are replaced by a state from $R(\mathcal{A}_1)$. The new automaton $\mathcal{A}_2$ is expecting word-like split-terms whose alphabet is augmented with $R(\mathcal{A}_1)$. A letter from $R(\mathcal{A}_1)$ only occurs on a leaf, and the transitions are augmented to assign the state s to a leaf labelled s for $s \in R(\mathcal{A}_1)$. Thus we obtain $\mathcal{A}_2$ over word-like split-terms, and $|\mathcal{A}_2| = \mathcal{O}(|\mathcal{A}_1|)$.

Hence for $\mathsf{BS}(k)$ we get complexity upper bounds as stated in Table 4.6 for all architectures, except for the problem of CPDS emptiness checking against fixed parameter. We also get complexity upper bounds as stated in Table 4.4 for bounded scope multi-pushdown systems, except for the problem of CPDS emptiness checking against fixed parameter. In the case of CPDS emptiness checking, our procedure is PTime due to the pre-computation phase. In fact we cannot do better since emptiness of nested-word automata is PTime-hard. Note that, for CPDS emptiness checking of bounded scope multi-pushdown systems with the bound as part of the input, we obtain a PSpace procedure which matches its known lower bound [LN11].

Note also that, if we obtain an exponential sized controller for $\mathsf{BS}(k)$, the complexities of the verification problems will be as stated in Table 4.4, except for CPDS emptiness checking against fixed parameter which would be PTime.

Conclusions Thus we may conclude that the class $\mathsf{BS}(k)$ also provides a reasonable class for the verification purposes. We have elementary decision procedures for all verification problems (except those involving MSO). This class also allows unbounded number of contexts which makes the case of infinite behaviours interesting.

This class also permits behaviours which are not existentially bounded and from cyclic architectures.

A very interesting problem which is left open here is to obtain a finite state controller for this class.

Chapter 13

Discussions

In this part of the thesis we have seen some classes of MSCNs, which allow under-approximate verification, and, at the same time are implementable.

However, this part is not an exhaustive listing of the classes. There are many other possibilities, and it is up to a researcher's imagination to come up with sensible classes to add into our 'controller library'. There are classes which could be defined independent of the framework of context graphs. There are classes which may jointly generalise a set of classes. We will briefly discuss some such classes now.

13.1 More results

13.1.1 Ordered multi-pushdown systems

A multi-pushdown system with ordering restriction [BCCCR96] assumes a total (priority) ordering on its stacks. It allows pops from a stack only if all the higher priority stacks are empty. However, there is no constraint on pushes. The reachability problem of such systems is decidable and is in fact 2-ETime-complete [ABH08]. The behaviours of such systems were shown to have a tree-width bounded by $\mathfrak{s}2^{\mathfrak{s}}$ in [MP11]. (Recall that $\mathfrak{s}$ denote the number of stacks.) Temporal logics model checking has been considered in [Ati10, LN12].

In [CGN12a] we show that the behaviours of ordered multi-pushdown systems have split-width bounded by $2^{\mathfrak{s}+1}$.[1] This class is easily MSO definable.

Multi-pushdown systems with ordering restriction is implementable by a CPDS whose number of states is at most exponential in the number of stacks. We will briefly describe this controller in the next paragraph. From the results of this thesis, it follows that this class has elementary (2-ExpTime) decision procedure for PDL model checking as well.

The controller for ordered multi-pushdown systems remembers in its state the subset of stacks which are non-empty. Thus pops are allowed only from

[1] From Theorem 5.2 this gives an improved bound on their tree-width.

the highest priority stack present in the current state. When it pushes to a non-empty stack, it retains its state. However, when it pushes to an empty stack s (one which is not present in its current state), it marks the value pushed into s with a special # symbol to indicate that it is the bottom-most symbol. Then it updates its state to include the stack s. Similarly, if it pops a # symbol from a stack s, it updates its state by removing stack s from the current state. Initial state as well as the final state of this controller is empty set ($\emptyset$). This deterministic controller is sound and complete for ordered multi-pushdown systems.

Since ordered-multi-pushdown systems admit an exponential bound on split-width as well as an exponential sized controller, the complexities of the various verification problems are as given in Table 4.5.

13.1.2 Joint generalisation of bounded phase / ordered with bounded scope

In [CGN12a] we also have other classes which jointly generalise bounded phase and bounded scope, or ordering and bounded scope[2]. The idea is that the pops try to follow the bounded scope restriction as much as possible, and those pops which fail to follow bounded scope must follow bounded phase or ordering restriction (when defining the phase/ordering we may discard the $\rhd$ edges which followed bounded scope). An example behaviour is given in Figure 13.1.

These classes are also shown to have bounded split-width, which is in fact the product of the split-width of the separate classes [CGN12b]. These classes are also implementable. The controller is essentially a cartesian product of the separate controllers. On a push it non-deterministically guesses whether the corresponding pop is going to follow bounded scope restriction or not, and this guess is also pushed into the stack, so that the guess can indeed be verified at the matching pop.

13.1.3 SR replaced by SW in $\mathsf{UASL}(k)$ and $\mathsf{DASL}(k)$

The condition SR can be replaced with a dual SW in the defintion of $\mathsf{UASL}(k)$ and $\mathsf{DASL}(k)$. The condition SW requires that a context may write to at most one data-structure. We may observe that the condition SR is the mirror of SW by reversing the direction of all edges. Thus the dual notions of $\mathsf{UASL}(k)$ and $\mathsf{DASL}(k)$ also form decidable classes with sound and complete controllers. The MSO definition, bound on split-width, the controller, and hence the complexities are all similar to the respective class ($\mathsf{UASL}(k)$ or $\mathsf{DASL}(k)$).

13.1.4 Existentially bounded MSCNs

An MSCN $\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \delta, \rightarrow, \rhd)$ is existentially bounded by k, if there is a linearisation of the events $\mathcal{E}$ such that the total usage of the data-structures

[2]A joint generalisation of bounded phase and ordered is Turing powerful.

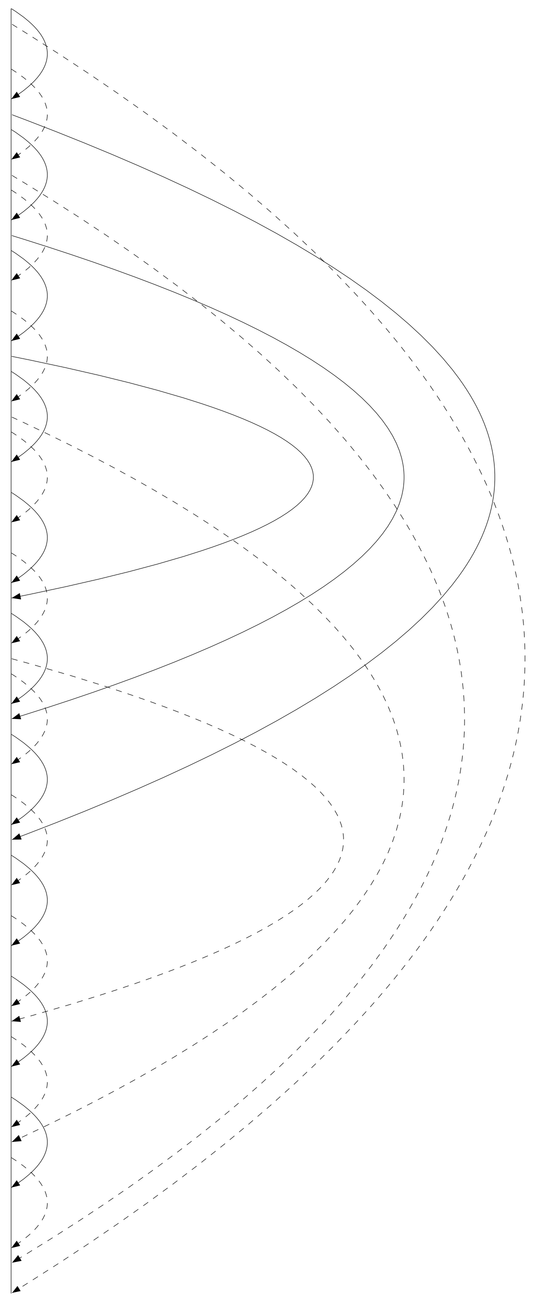


Figure 13.1: A multiply nested word following scope-or-phase-bounded policy. The bound on scope is 3 whereas it has only 2 phases. This behaviour follows scope-bounded-or-ordered policy as well. The solid stack has higher priority than the dashed stack.

does not exceed k. Let $<_{\text{lin}}$ denote the linearisation. MSCN $\mathcal{M}$ is *existentially k bounded* if for every event e, $|\{e' \mid e' \rhd f' \text{ and } e' \leq_{\text{lin}} e <_{\text{lin}} f'\}| \leq k$. We denote the set of existentially k bounded MSCNs by $\mathsf{EB}(k)$. $\mathsf{EB}(k)$

For the classes $\mathsf{EB}(k)$ and a generalisation $\mathsf{GEB}(k)$ (Section 13.1.5) we give a bound on split-width. But we leave it for future work to investigate controllers and to obtain MSO formulas for these classes. The challenge would be to assume and ensure a linearisation respecting the bound k.

In [GKM06], it is shown that the class of existentially bounded MSCs (without stacks) is definable in MSO and admits a distributed implementation. This would be a possible starting point to study the generalisation towards MSCNs.

Theorem 13.1. *If an MSCN $\mathcal{M}$ is in $\mathsf{EB}(k)$, then the split-width of $\mathcal{M}$ is at most $k+1$. Moreover, it is word-like.*

Proof idea. To obtain a split-decomposition we detach events (that is, make the $\rightarrow$ edges incident on it into $\overset{e}{\rightarrow}$ edges) one by one in the order of the linearisation as singleton contexts. After an event is detached we may remove it (together with its $\rhd$ partner if it is available as a singleton context) so that the following invariant is kept.

1. All the unremoved detached events are writes.

2. There are at most k unremoved detached events

Thus, if the current detached event is an internal event, it is removed from the rest as a child of shuffle. If it is a write event we keep it as a separate component of the split-MSCN. If it is a read event, then the matching write is already present as a singleton context. Hence this $\rhd$ edge can be removed. Note that all the singleton contexts are write events. Moreover since it is existentially k bounded and the events are detached according to the linearisation, the number of unremoved detached contexts cannot exceed k.

The detached contexts are the only possible sources of an elastic edge $\overset{e}{\rightarrow}$, hence the split width is bounded by the maximal number of detached contexts possible at any instant, which is $k+1$. There can be $k+1$ detached contexts when there are already k write events detached as single contexts, and on detaching the subsequent event (which would be a read or an internal, and hence will be removed to satisfy the invariants).

Thus for obtaining a split-term, we are treating MSCNs in the order of the linearisation (a single left-to-right scan). Only one event is processed at a time. Depending on its type, either it stays as a singleton context, or is detached from the main spine of the split-term as the child of a shuffle node. Clearly this gives a word-like split-term. □

13.1.5 Generalisation of existentially bounded MSCNs

We can consider a much richer class of MSCNs inspired by the above result on existentially bounded MSCNs. Let $\mathsf{EB}(k)$ be the set of all MSCNs with an existential bound of k. An MSCN is generalised existentially bounded by k if it admits a context-decomposition which looks like an MSCN in $\mathsf{EB}(k)$. For this we describe how to extract out a possibly-MSCN-like structure from a context decomposition.

Let $\mathcal{M}\!\!\!/$ be a split-MSCN with $G_{\mathcal{M}\!\!\!/} = (V, (E_d)_{d \in \mathbf{DS}})$ its extended context graph. The structure $\mathsf{Quotient}(\mathcal{M}\!\!\!/) = (\mathcal{E}, \lambda, \mathsf{pid}, \delta, \rightarrow, \rhd)$ where

$\mathsf{Quotient}(\mathcal{M}\!\!\!/)$

- $\mathcal{E} = V$
- $\lambda(v) = a$
- $\mathsf{pid}(v = (p, i)) = p$
- $\delta(v = (p, i)) = \{d \mid \delta(e) = d \text{ for some } e \in v\}$
- $\rightarrow = \{(v, v') \mid v = (p, i) \text{ and } v' = (p, i + 1)\}$
- $\rhd^d = E_d$

We say a split-MSCN $\mathcal{M}\!\!\!/$ looks like an MSCN if $\mathsf{Quotient}(\mathcal{M}\!\!\!/)$ is an MSCN over $\Sigma = \{a\}$. This requires that a context can be part of at most one edge from $\cup E_d$. Hence not only a context accesses at most one data-structure in one direction, but also all the $\rhd$ edges from/to this context are to/from a single other context. In particular, stack loops are forbidden.

We say an MSCN $\mathcal{M}$ is in $\mathsf{GEB}(k)$, if it admits a context decomposition $\mathcal{M}\!\!\!/$ such that $\mathsf{Quotient}(\mathcal{M}\!\!\!/) \in \mathsf{EB}(k)$.

$\mathsf{GEB}(k)$

Theorem 13.2. *If an MSCN $\mathcal{M}$ is in $\mathsf{GEB}(k)$, then the split-width of $\mathcal{M}$ is at most $k + 3$. Moreover, it is word-like.*

Proof idea. Let $\mathcal{M}\!\!\!/$ be a witnessing context decomposition with a linearisation. The idea is to simulate the proof of bounded split-width of $\mathsf{EB}(k)$ on $\mathsf{Quotient}(\mathcal{M}\!\!\!/)$. The components are extracted out in the order of the linearisation. These components need not be singleton, but will be accessing at most one data-structure and in at most one direction (either write or read, even for stacks). If the currently extracted context is a "write" context, it stays as a context. However, we may trim this context by removing the brinks that are internal. If the currectly extracted context is not accessing a data-structure, then we remove the brinks one by one until this context disappears. If the currently extracted context is a "read" context, then brinks of this read context are linked to the brinks of an already extracted context. Hence we may remove the $\rhd$ edges connecting brinks one by one, and also the brinks which are internal, as soon as they appear, until the two contexts disappear. □

13.2 Impact of the results

The impacts of our results can be best appreciated on comparison with existing related works. Please refer to Section 1.2 for a brief overview of related works.

Reachability problems over specific architectures have been an important topic of study in formal verification. Temporal logics and model checking have also been studied for some architectures. Various restrictions have been proposed to obtain decidability. Decidability is shown for each of these restrictions using ad-hoc techniques[3].

uniform decision procedure

On the contrary, split-width offers a generic proof technique, not only for various restrictions, but also for various verification problems. Moreover, it gives the best known complexities in most cases.

various restrictions

various verification problems

optimal complexity

The approach for bounding the split-width is also generic and systematic. This easily extends to generalisations as well.

easy to extend

In essence, split-width is a new technique for the verification of CPDS. It gives a parameter for the under-approximate verification, and a useful encoding of the behaviours as trees. It also provides alternate proofs for the existing decidability results. In fact we have been able to find new decidable classes, thanks to split-width technique. It is worth mentioning that even on specific architectures well-studied in the literature, our results throw new lights.

13.3 Perspectives

It is left for future work to extend the classes discussed in Section 13.1 for multi-pushdown systems to the more general setting of MSCNs, as well as investigating good controllers for them.

Another direction of research is to tackle different desirable features of the controllers. Is it possible to get a deterministic controller for a class? Can we get a controller which is deadlock-free? Is it possible to gain one feature by compromising another? For example, can we get a deadlock-free controller by surrendering the finite state property and settling for some time stamping like protocols (cf. controller for $\mathsf{BS}(k)$)? This would be a very promising direction of research with many interesting problems that need to be answered.

This part also unveils the beauty and power of split-width. It shows us how split-width explores the 'structure' of a class, and enlightens our understanding of a class of restrictions of MSCNs. It also exemplifies the divide-and-conquer approach for showing the split-width.

[3][MP11] gives a unifying proof of decidability of reachability for some restrictions.

Chapter 14

Conclusions

Summary The manuscript introduces the notion of split-width as a tool for the verification of concurrent processes with data-structures. It illustrates a convenient tree-representation for the behaviours with bounded split-width. Then it demonstrates how to make use of the rich results and techniques available for trees to aid the verification of CPDS. It provides uniform decision procedures for various verification problems. The complexity of the uniform decision procedures are optimal for most of these problems.

It proceeds to emphasise the power of split-width. In Chapter 5 we have seen that split-width can capture all bounded clique-width behaviour graphs. Thus it follows that it can capture any class of behaviours with a decidable MSO theory.

In another direction it illustrates the power of split-width by considering several natural restrictions on CPDS and showing that these have bounded split-width. These classes are general and extends several known decidable restrictions. Moreover, these classes are implementable by distributed controllers.

The thesis discusses the necessity and advantages of distributed controllers. It proposes some desirable features to keep in mind while designing one. Distributed controllers for several natural classes are given, and these are shown to be sound and complete.

Future work We have stated interesting open questions and further directions research throughout the manuscript. A very important one of these is, given a regular language over tree-representations of MSCNs of split-width at most k, can we obtain a CPDS recognising the represented MSCNs (cf. Open Question 6.8). If we can answer this positively, then the split-width technique can be used for complementing a CPDS wrt. a bound on split-width. This would also imply that various controllable classes are complementable.

There are several classes for which we need to investigate whether they admit reasonable controllers. It is interesting to see whether the class $\mathsf{BS}(k)$ admits a finite state controller, or whether the acyclic classes admit a deterministic controller. Another direction that ought to be taken is investigating the controllers to have further desirable properties like determinism, dead-lock freedom etc.

Further directions From a wider perspective, the theory of split-width may be extended along several dimensions. For example, to handle infinite behaviours. The challenge is that, the current definition of split-width resonates with a least fixed point, which does not naturally lift up to infinite behaviours. Another possibility is to extend this theory to handle infinite data (cf. [BCGK12]). Yet another dimension is broadening the application domain, e.g. allowing data-structures other than stacks and queues. Note that the notion of split-width is defined for CBMs which may be thought of as handling bag data-structures. However, we do not know natural classes of CPDS with bag data-structures that would admit a bound on split-width. In the thesis we have only considered linear time properties of CPDS. Applications of split-width to the verification of branching time properties is another direction.

It is interesting to see whether split-width can be defined when a linear order of events along a process is replaced by a partial order of events (cf. nested traces [BGH09]).

A lot of algorithmic questions can also be asked in the case of split-width, e.g. obtaining the optimal split-term for an input MSCN.

Bibliography

[AAB$^+$08] R. Alur, M. Arenas, P. Barceló, K. Etessami, N. Immerman, and L. Libkin. First-order and temporal logics for nested words. *Log. Meth. Comput. Sci.*, 4(4), 2008. 11, 39, 81

[Abd10] P. A. Abdulla. Forcing monotonicity in parameterized verification: From multisets to words. In J. van Leeuwen, A. Muscholl, D. Peleg, J. Pokorný, and B. Rumpe, editors, *SOFSEM'10*, volume 5901 of *LNCS*, pages 1–15. Springer, 2010. 8

[ABH08] Mohamed Faouzi Atig, Benedikt Bollig, and Peter Habermehl. Emptiness of multi-pushdown automata is 2ETIME-Complete. In Masami Ito and Masafumi Toyama, editors, *Developments in Language Theory*, volume 5257, pages 121–133. Springer, 2008. 5, 6, 46, 169

[ABKS12] Mohamed Faouzi Atig, Ahmed Bouajjani, K. Narayan Kumar, and Prakash Saivasan. Linear-time model-checking for multithreaded programs under scope-bounding. In Supratik Chakraborty and Madhavan Mukund, editors, *ATVA*, volume 7561 of *Lecture Notes in Computer Science*, pages 152–166. Springer, 2012. 7

[ABQ11] M. F. Atig, A. Bouajjani, and S. Qadeer. Context-bounded analysis for concurrent programs with dynamic creation of threads. *Logical Methods in Computer Science*, 7(4), 2011. 14

[AEM04] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In K. Jensen and A. Podelski, editors, *TACAS 2004*, volume 2988 of *LNCS*, pages 467–481. Springer, 2004. 11, 39

[AJ96] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. *Information and Computation*, 127(2):91 – 101, 1996. 8, 46

[AKMV05] Rajeev Alur, Viraj Kumar, P. Madhusudan, and Mahesh Viswanathan. Congruences for visibly pushdown languages. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *ICALP*, volume 3580 of *Lecture Notes in Computer Science*, pages 1102–1114. Springer, 2005. 50

[AKS13] Mohamed Faouzi Atig, K. Narayan Kumar, and Prakash Saivasan. Adjacent ordered multi-pushdown systems. In Marie-Pierre Béal and Olivier Carton, editors, *Developments in Language Theory*, volume 7907 of *Lecture Notes in Computer Science*, pages 58–69. Springer, 2013. 5, 6

[AM09] R. Alur and P. Madhusudan. Adding nesting structure to words. *Journal of the ACM*, 56(3):1–43, 2009. 4, 10, 11, 25, 39, 50, 81

[APP95] Rajeev Alur, Doron Peled, and Wojciech Penczek. Model-checking of causality properties. In *LICS*, pages 90–100. IEEE Computer Society, 1995. 36

[Ati10] Mohamed Faouzi Atig. From multi to single stack automata. In Paul Gastin and François Laroussinie, editors, *CONCUR*, volume 6269, pages 117–131. Springer, 2010. 169

[BCCCR96] Luca Breveglieri, Alessandra Cherubini, Claudio Citrini, and Stefano Crespi-Reghizzi. Multi-pushdown languages and grammars. *Int. J. Found. Comput. Sci.*, 7(3):253–292, 1996. 5, 6, 46, 169

[BCGK12] Benedikt Bollig, Aiswarya Cyriac, Paul Gastin, and K. Narayan Kumar. Model checking languages of data words. In *FoSSaCS*, volume 7213 of *Lecture Notes in Computer Science*, pages 391–405. Springer, 2012. 12, 176

[BCGZ11] Benedikt. Bollig, Aiswarya. Cyriac, Paul. Gastin, and Marc. Zeitoun. Temporal logics for concurrent recursive programs: Satisfiability and model checking. In *MFCS*, volume 6907 of *Lecture Notes in Computer Science*, pages 132–144. Springer, 2011. 7, 11, 38, 39, 40, 77, 80, 81, 148

[BCH$^+$13] Benedikt Bollig, Aiswarya Cyriac, Loïc Hélouët, Ahmet Kara, and Thomas Schwentick. Dynamic communicating automata and branching high-level MSCs. In *LATA*, volume 7810 of *Lecture Notes in Computer Science*, pages 177–189, Bilbao, Spain, April 2013. Springer. 13

[BDM$^+$11] M. Bojańczyk, C. David, A. Muscholl, Th. Schwentick, and L. Segoufin. Two-variable logic on data words. *ACM Trans. Comput. Log.*, 12(4):27, 2011. 12

[BGH09] B. Bollig, M.-L. Grindei, and P. Habermehl. Realizability of concurrent recursive programs. In L. de Alfaro, editor, *FOSSACS 2009*, volume 5504 of *LNCS*, pages 410–424. Springer, 2009. 11, 18, 176

[BGP08] J. Borgström, A. Gordon, and A. Phillips. A chart semantics for the Pi-calculus. *Electronic Notes in Theoretical Computer Science*, 194(2):3–29, 2008. 14

[BH10] B. Bollig and L. Hélouët. Realizability of dynamic MSC languages. In *Proceedings of CSR'10*, volume 6072 of *LNCS*, pages 48–59. Springer, 2010. 14

[BKM10] B. Bollig, D. Kuske, and I. Meinecke. Propositional dynamic logic for message-passing systems. *Logical Methods in Computer Science*, 6(3:16), 2010. 9, 34, 39

[BKM13] Benedikt Bollig, Dietrich Kuske, and Roy Mennicke. The complexity of model checking multi-stack systems. *Logic in Computer Science, Symposium on*, 0:163–172, 2013. 7

[BL06] B. Bollig and M. Leucker. Message-passing automata are expressively equivalent to EMSO logic. *Theoretical Computer Science*, 358(2):150–172, 2006. 31, 45

[BLP08] L. Bozzelli, S. La Torre, and A. Peron. Verification of well-formed communicating recursive state machines. *Theoretical Computer Science*, 403(2-3):382–405, 2008. 14

[Bol05] Benedikt Bollig. *Automata and Logics for Message Sequence Charts*. Thèse de doctorat, Department of Computer Science, RWTH Aachen, Germany, May 2005. 31

[Bol08] Benedikt Bollig. On the expressive power of 2-stack visibly pushdown automata. *Logical Methods in Computer Science*, 4(4:16), December 2008. 45

[Bol11] B. Bollig. An automaton over data words that captures EMSO logic. In J.-P. Katoen and B. König, editors, *CONCUR'11*, volume 6901 of *LNCS*, pages 171–186. Springer, 2011. 12

[BS01] M. G. Buscemi and V. Sassone. High-level Petri nets as type theories in the join calculus. In *Proceedings of FOSSACS'01*, volume 2030 of *LNCS*, pages 104–120. Springer, 2001. 14

[BZ83] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2), 1983. 5, 8, 39, 44

[CE95] Bruno Courcelle and Joost Engelfriet. A logical characterization of the sets of hypergraphs defined by hyperedge replacement grammars. *Mathematical Systems Theory*, 28(6):515–552, 1995. 90

[CE12] Bruno Courcelle and Joost Engelfriet. *Graph Structure and Monadic Second-Order Logic - A Language-Theoretic Approach*, volume 138 of *Encyclopedia of mathematics and its applications.* Cambridge University Press, 2012. 89, 93

[CGN12a] Aiswarya Cyriac, Paul Gastin, and K. Narayan Kumar. MSO decidability of multi-pushdown systems via split-width. In *CONCUR*, volume 7454 of *Lecture Notes in Computer Science*, pages 547–561. Springer, 2012. 3, 7, 80, 81, 105, 169, 170

[CGN12b] Aiswarya Cyriac, Paul Gastin, and K. Narayan Kumar. MSO decidability of multi-pushdown systems via split-width. Research Report LSV-12-11, Laboratoire Spécification et Vérification, ENS Cachan, France, September 2012. 36 pages. 170

[Cha11] Pierre Chambart. *Du Problème de sous-mot de Post et de la complexité des canaux non fiables.* Thèse de doctorat, Laboratoire Spécification et Vérification, ENS Cachan, France, September 2011. 44, 46

[CK98] Edward Y. C. Cheng and Michael Kaminski. Context-free languages over infinite alphabets. *Acta Inf.*, 35(3):245–267, 1998. 12

[CO00] Bruno Courcelle and Stephan Olariu. Upper bounds to the clique width of graphs. *Discrete Applied Mathematics*, 101(1-3):77–114, 2000. 93

[Cou97] Bruno Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In Grzegorz Rozenberg, editor, *Handbook of Graph Grammars*, pages 313–400. World Scientific, 1997. 8

[Cou06] Bruno Courcelle. The monadic second-order logic of graphs XV: On a conjecture by D. Seese. *Journal of Applied Logic*, 8:1–40, 2006. 90

[Cou10] Bruno Courcelle. Special tree-width and the verification of monadic second-order graph pr operties. In *FSTTCS*, volume 8 of *LIPIcs*, pages 13–29, 2010. 91, 93

[CR01] Derek G. Corneil and Udi Rotics. On the relationship between clique-width and treewidth. In Andreas Brandstädt and Van Bang Le, editors, *WG*, volume 2204 of *Lecture Notes in Computer Science*, pages 78–90. Springer, 2001. 93

[Cyr10] Aiswarya Cyriac. Temporal logics for concurrent recursive programs. Master's thesis, Master Parisien de Recherche en Informatique, Paris, France, September 2010. 50

[Cyr12] Aiswarya Cyriac. Model Checking Dynamic Distributed Systems. In *Winter School Modelling and Verifying Parallel processes (MOVEP'12)*, Marseille, France, December 2012. 12

[DG06] V. Diekert and P. Gastin. Pure future local temporal logics are expressively complete for Mazurkiewicz traces. *Information and Computation*, 204(11):1597–1619, 2006. 10, 11, 36

[DK11] C. Dax and F. Klaedtke. Alternation elimination for automata over nested words. In M. Hofmann, editor, *FOSSACS 2011*, LNCS, pages 168–183. Springer, 2011. 11

[DLS08] S. Demri, R. Lazić, and A. Sangnier. Model checking freeze LTL over one-counter automata. In R. M. Amadio, editor, *FoSSaCS'08*, volume 4962 of *LNCS*, pages 490–504. Springer, 2008. 12

[DR95] V. Diekert and G. Rozenberg, editors. *The Book of Traces*. World Scientific, Singapore, 1995. 10

[DS10] S. Demri and A. Sangnier. When model-checking freeze LTL over counter machines becomes decidable. In C.-H. L. Ong, editor, *FoSSaCS'10*, volume 6014 of *LNCS*. Springer, 2010. 12

[FL79] M.J. Fischer and R.E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979. 11, 75

[FS01] Alain Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001. 8

[GK03] P. Gastin and D. Kuske. Satisfiability and model checking for MSO-definable temporal logics are in PSPACE. In R.M. Amadio and D. Lugiez, editors, *CONCUR'03*, volume 2761 of *LNCS*, pages 222–236. Springer, 2003. 11, 36

[GK10] P. Gastin and D. Kuske. Uniform satisfiability problem for local temporal logics over Mazurkiewicz traces. *Information and Computation*, 208(7):797–816, 2010. 11, 36, 38, 39, 77

[GKM06] B. Genest, D. Kuske, and A. Muscholl. A Kleene theorem and model checking algorithms for existentially bounded communicating automata. *Information and Computation*, 204(6):920–956, 2006. 5, 46, 172

[GKM07] Blaise Genest, Dietrich Kuske, and Anca Muscholl. On communicating automata with bounded channels. *Fundam. Inform.*, 80(1-3):147–167, 2007. 5, 8

[GKS10] O. Grumberg, O. Kupferman, and S. Sheinvald. Variable automata over infinite alphabets. In A. H. Dediu, H. Fernau, and C. Martín-Vide, editors, *LATA'10*, volume 6031 of *LNCS*, pages 561–572. Springer, 2010. 12

[GLL09] Stefan Göller, Markus Lohrey, and Carsten Lutz. Pdl with intersection and converse: satisfiability and infinite-state model checking. *J. Symb. Log.*, 74(1):279–314, 2009. 76

[HLMS10] Alexander Heußner, Jérôme Leroux, Anca Muscholl, and Grégoire Sutre. Reachability analysis of communicating pushdown systems. In C.-H. Luke Ong, editor, *FOSSACS*, volume 6014, pages 267–281. Springer, 2010. 8, 40, 44, 45, 142

[HMN$^+$05] J. G. Henriksen, M. Mukund, K. Narayan Kumar, M. A. Sohoni, and P. S. Thiagarajan. A theory of regular MSC languages. *Inf. Comput.*, 202(1):1–38, 2005. 5, 46

[IT11] ITU-TS. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC).* ITU-TS, Geneva, February 2011. 4, 8, 18, 24

[Kam68] H. Kamp. *Tense Logic and the Theory of Linear Order.* PhD thesis, University of California, Los Angeles, 1968. 10

[KF94] M. Kaminski and N. Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994. 12, 14

[Kre09] Stephan Kreutzer. Algorithmic meta-theorems. *CoRR*, abs/0902.3616, 2009. 8

[KZ10] M. Kaminski and D. Zeitlin. Finite-memory automata with non-deterministic reassignment. *Int. J. Found. Comput. Sci.*, 21(5):741–760, 2010. 12

[Lib06] Leonid Libkin. Logics for Unranked Trees: An Overview. *Log. Meth. Comput. Sci.*, 2(3), 2006. 11

[LMM02] Martin Leucker, P. Madhusudan, and Supratik Mukhopadhyay. Dynamic message sequence charts. In *FSTTCS*, volume 2556 of *Lecture Notes in Computer Science*, pages 253–264. Springer, 2002. 14

[LMP07] Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. A robust class of context-sensitive languages. In *LICS*, pages 161–170. IEEE Computer Society, 2007. 5, 6, 11, 46, 80, 81, 143, 148

[LMP08a] S. La Torre, P. Madhusudan, and G. Parlato. Context-bounded analysis of concurrent queue systems. In *TACAS'08*, volume 4963 of *LNCS*, pages 299–314. Springer, 2008. 6, 8, 33, 40, 45, 84, 136, 142, 148

[LMP08b] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. An infinite automaton characterization of double exponential time. In *CSL*, volume 5213 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2008. 81, 148

[LN11] Salvatore La Torre and Margherita Napoli. Reachability of multi-stack pushdown systems with scope-bounded matching relations. In Joost-Pieter Katoen and Barbara König, editors, *CONCUR*, volume 6901, pages 203–218. Springer, 2011. 5, 6, 46, 150, 167

[LN12] Salvatore La Torre and Margherita Napoli. A temporal logic for multi-threaded programs. In Jos C. M. Baeten, Thomas Ball, and Frank S. de Boer, editors, *IFIP TCS*, volume 7604 of *Lecture Notes in Computer Science*, pages 225–239. Springer, 2012. 7, 40, 150, 169

[LP12] Salvatore La Torre and Gennaro Parlato. Scope-bounded multi-stack pushdown systems: fixed-point, sequentialization, and tree-width. Technical report, University of Southampton, February 2012. 7, 8, 150

[LR09] Akash Lal and Thomas W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, 2009. 6

[LTKR08] Akash Lal, Tayssir Touili, Nicholas Kidd, and Thomas W. Reps. Interprocedural analysis of concurrent programs under a context bound. In Ramakrishnan and Rehof [RR08], pages 282–298. 6

[LW00] K. Lodaya and P. Weil. Series-parallel languages and the bounded-width property. *Theoretical Computer Science*, 237(1-2):347 – 380, 2000. 14

[LW01] K. Lodaya and P. Weil. Rationality in algebras with a series operation. *Information and Computation*, 171(2):269 – 293, 2001. 14

[Men13a] Roy Mennicke. Model Checking Concurrent Recursive Programs, 2013. 77

[Men13b] Roy Mennicke. Propositional dynamic logic with converse and repeat for message-passing systems. *Logical Methods in Computer Science*, 9(2), 2013. 9

[Mey08] R. Meyer. On boundedness in depth in the π-calculus. In *Proceedings of IFIP TCS'08*, volume 273 of *IFIP*, pages 477–489. Springer, 2008. 14

[MP11] P. Madhusudan and Gennaro Parlato. The tree width of auxiliary storage. In Thomas Ball and Mooly Sagiv, editors, *POPL*, pages 283–294. ACM, 2011. 7, 8, 40, 46, 136, 169, 174

[MR04] B. Meenakshi and Ramaswamy Ramanujam. Reasoning about layered message passing systems. *Computer Languages, Systems & Structures*, 30(3-4):171–206, 2004. 9

[Nar12] K. Narayan Kumar. The theory of MSC languages. In *Modern Applications of Automata Theory*, pages 289–324. World Scientific, 2012. 8

[Pel00] Doron Peled. Specification and verification of message sequence charts. In *FORTE*, volume 183 of *IFIP Conference Proceedings*, pages 139–154. Kluwer, 2000. 9

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of FOCS 1977*, pages 46–57. IEEE, 1977. 10

[QR05] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In Nicolas Halbwachs and Lenore D. Zuck, editors, *TACAS*, volume 3440, pages 93–107. Springer, 2005. 5, 6, 46, 138

[RR08] C. R. Ramakrishnan and Jakob Rehof, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963. Springer, 2008. 183

[Sch02] Philippe Schnoebelen. Verifying lossy channel systems has nonprimitive recursive complexity. *Information Processing Letters*, 83(5):251–261, September 2002. 46

[See91] Detlef Seese. The structure of models of decidable monadic theories of graphs. *Ann. Pure Appl. Logic*, 53(2):169–195, 1991. 8, 90

[Thi94] P. S. Thiagarajan. A trace based extension of linear time temporal logic. In *LICS*, pages 438–447. IEEE Computer Society, 1994. 36, 39

[Tze11] N. Tzevelekos. Fresh-register automata. In Th. Ball and M. Sagiv, editors, *POPL'11*, pages 295–306. ACM, 2011. 12

[Var85] M. Y. Vardi. The taming of converse: Reasoning about two-way computations. In *Proc. of the Conference on Logic of Programs*, pages 413–423. Springer, 1985. 66, 75

[Var98] M. Y. Vardi. Reasoning about the past with two-way automata. In K.G. Larsen, S. Skyum, and G. Winskel, editors, *ICALP'98*, LNCS, pages 628–641. Springer, 1998. 65, 66, 76

[Zie87] W. Zielonka. Notes on finite asynchronous automata. *R.A.I.R.O. — Informatique Théorique et Applications*, 21:99–135, 1987. 10

Index