

Automata over Nested Data



Adriana Baldacchino
St Catherine's College
University of Oxford

A thesis submitted for the degree of
MSc. Mathematics and Foundations of Computer Science

Trinity 2025

Acknowledgements

Firstly I wish to thank my supervisor, Professor Andrzej Murawski for all his patience and support throughout this dissertation, and for encouraging me to present my work. I would also like to thank my MFoCS coursemates for their company and encouragement for the past year, and the administration for all their help. A huge thanks goes to my parents, Romina and Adrian — without them, I would not have had this lovely opportunity. Lastly, I would like to thank my partner and best friend Xandru for always cheering me on.

Abstract

Automata over nested data are automata over an infinite alphabet, where the underlying infinite set is additionally equipped with a hierarchical structure. Recently, there has been extensive research on automata over infinite alphabets due to their applications in computer science, from databases to verification. A model of automata over nested data — *nested data class memory automata (NDCMA)* has been applied in the study of programming languages. However, all existing models of automata over nested data are restricted to only handling bounded depth data, where the hierarchical structure is limited. This constraint limits the application of NDCMA to only programs without loops.

We extend NDCMA to handle unbounded data by introducing two new models of automata over nested data, *flexible NDCMA* and *rigid NDCMA*. Using the theory of well-structured transition systems, we prove that these new models have a decidable emptiness problem. We further explore closure properties of these automata, such as closure under union, intersection and complement, and decidability of inclusion and equivalence. Finally, we provide an application for rigid NDCMA in representing the trace semantics of a program with loops.

Contents

1	Introduction	1
1.1	Automata over Infinite Alphabets	1
1.2	Automata over Nested Data	2
1.3	Contributions	4
2	Well-quasi orders and WSTS	6
2.1	Well-quasi orders	6
2.1.1	Combining well-quasi orders	8
2.1.2	Well-quasi orders on trees	8
2.2	Well-Structured Transition Systems	13
2.2.1	Decision problems on WSTS	14
3	Flexible NDCMA	16
3.1	Introducing f-NDCMA	16
3.2	WSTS for f-NDCMA	20
3.3	Closure properties	25
4	Rigid NDCMA	29
4.1	Introducing r-NDCMA	29
4.2	WSTS for r-NDCMA	34
4.3	Closure Properties	40
4.4	Petri Nets and Simulation	50
5	Applying Unbounded NDCMA	59
5.1	Language Syntax and Semantics	59
5.2	Trace semantics	62

5.3	Replacing O-names with indices	69
5.4	Translation to NDCMA	74
5.4.1	Representing plays	74
5.4.2	Handling the tree structure	75
5.5	Further work and limitations	78
Bibliography		80

Chapter 1

Introduction

1.1 Automata over Infinite Alphabets

Automata over infinite alphabets are widely studied for their applications in computer science. These automata extend classical automata by considering *data words*, which are words in which each position has a *tag* from some finite set, and a *data value* from an infinite set. This infinite set is useful in modelling various unbounded sets that arise naturally in computer science, such as data values in database theory [8, 22, 16], unbounded resources in verification [18, 11] and names in programming language semantics [13, 9].

Since these automata are used for vastly different applications, it follows that different models for automata over infinite alphabets were developed over the years. We give a brief overview of some popular models — more detail can be found in extensive surveys on automata (and logics) over infinite alphabets [10, 36]. One of the first models introduced was *register automata* (sometimes called *finite memory automata*) [23]. These automata make use of a finite number of registers in which data values can be stored and compared. In a similar vein, *pebble automata* [28, 29] navigate the data word, placing some bounded number of pebbles on the word and testing equality on these pebbled symbols. An alternative discipline for automata over infinite alphabets is through *atoms* and *orbit-finite sets* [6, 7, 4]. In these works, the underlying infinite set is equipped with symmetries that allow it to be processed in a finite way. This can be used to synthesize objects with the same expressive power as register and pebble automata [6].

Another model for automata, *data automata* was introduced in [5]. This model was introduced to show decidability of two variable logic over data words, and is in fact equivalently expressive as existential monadic second order logic over data words, $\text{EMSO}^2(<, +1, \sim)$. These automata consist of a base automaton and a class automaton, and process a word in two stages. First, the base automaton takes the projection of the data word over the finite alphabet and returns a word over a finite alphabet. Then, this word is partitioned into subwords based on the data values, and these words are further checked using the class automaton. This model is quite expressive, but somewhat challenging to use because of the complicated definition. In [3], the authors introduce an equivalent model of automata, *class memory automata (CMA)*, which have a more intuitive definition. The simple idea behind this model is that whenever the automaton encounters a data value, it has access to the state in which the data value was last seen, and moreover can tell if the data value is new.

Class memory automata are not closed under complement, even if we consider the deterministic versions of these automata. To amend this, a subclass of these automata, *weak class memory automata* were introduced in [14]. In this work, deterministic weak class memory automata were shown to be closed under complement. Moreover, they were shown to be equivalent to prefix-closed data automata (pDA) as introduced in [15], which are data automata for which all the states of the class automaton are accepting. Furthermore, it was shown that these weak class memory automata are equiexpressive with the bounded fragment of $\text{EMSO}^2(<, +1, \sim)$ [12].

1.2 Automata over Nested Data

Nested data as introduced in [2] is data with additional hierarchical structure. In [2, 15], this was presented by considering words where each position of the word consists of the finite tag, and a *tuple* of data values of some fixed size k . The hierarchical structure here arises in the equivalence relations $\sim_n$ considered, which hold for tuples $(d_1, \dots, d_k), (d'_1, \dots, d'_k)$, if and only if the first n entries are equal. An alternate presentation for nested data was introduced in [14], where the data set has an implicit underlying forest structure. In this work, it was also shown

that this presentation is equivalent to the tuple presentation, as there are effective translations between the two.

Logics over nested data quickly become undecidable. Even with only two layers of nested data, two-variable first-order logic equipped with $x = y + 1$ and $x < y$ is undecidable. This was shown by a translation to multicounter automata [2]. In this same paper, it was shown that restricting to only the successor function results in a decidable logic, by a translation to shuffle expressions. A similar positive result was established in [15], where the authors established ND-LTL, a temporal logic on nested data words extending LTL by navigation along data values, which has a decidable satisfiability question. To do this, they introduce *nested data automata (NDA)*, which are automata over nested data. An equiexpressive model of automata, *nested data class memory automata (NDCMA)* was defined in [14]. This model was defined on the forest representation of nested data, and works similarly to class memory automata. For NDCMA, when a data value is accessed, the automaton has access to not only the state in which the data value was last seen, but also to the states in which its ancestors (according to the forest structure) were last seen. This extends the behaviour of normal class memory automata.

As is true for classical CMA and data automata, we can define weak and prefix-closed versions (respectively) of these nested data automata, and moreover these coincide. It was also shown that these models are closed under union and intersection, and have a decidable emptiness problem. Since deterministic weak NDCMA are furthermore closed under complement, equivalence of these automata is decidable. This means that if we can represent objects using deterministic weak NDCMA, we could deduce that their equivalence problem is decidable.

Due to their nice closure properties and ability to represent nested data values, NDCMA in particular have been applied to study a fragment of ML with first-order state (FOS) [13, 9]. This model was initially applied to capture the game semantics of FOS with very limited types [13] (and with respect to FOS contexts), and prove decidability of the associated contextual equivalence problem. This was further developed by Bunting and Murawski to allow for continuations and unrestricted types in terms (FOSC) and contexts with both continuations and higher-order state (HOSC) [9], a scenario they referred to as HOSC[FOSC].

A key limitation to all the existing models of automata over nested data is that the nesting is limited by some bound k . In the tuple representation, this is because each tuple is of size k , and for the forest representation this limitation arises as a bound for the depth of the forest structure of the data set. Therefore, an interesting question to ask is whether this limitation can be lifted without losing any closure and decidability properties of the automata. Moreover, this limitation also arises in the applications of NDCMA as discussed above, as the bounded nature of NDCMA means that we can only consider some bounded number of moves dependent on previous ones. Practically speaking, this means that the languages considered have to be loop free, as adding loops leads to a potentially unbounded sequence of dependent moves. For this reason, we explore new models for unbounded nested data with the aim of extending the existing results for programs with loops.

1.3 Contributions

This work revolves around extending NDCMA for unbounded data, and hence contains some original results. These include:

- Introduction of two new models of NDCMA for unbounded data — flexible NDCMA (f-NDCMA) and rigid NDCMA (r-NDCMA).
- Closure properties for f-NDCMA:
 - Proof of decidable emptiness of weak f-NDCMA, using WSTS techniques.
 - Proof of closure under union and intersection.
 - Explanation as to why these automata cannot be made suitably deterministic.
- Closure properties of r-NDCMA:
 - Proof of decidable emptiness of weak r-NDCMA, using WSTS techniques.
 - Introduction of deterministic weak r-NDCMA, which are closed under complement.

- Proof that the problem of deciding emptiness of the intersection of deterministic weak r-NDCMA is undecidable.
- Utilisation of the above result to show that deterministic weak r-NDCMA are not effectively closed under intersection and union, inclusion is undecidable, and for non-deterministic weak r-NDCMA, equivalence is undecidable.
- Equivalence of deterministic prefix reset Petri net languages was shown to be undecidable. We further show that this equivalence can be reduced to equivalence of the language projection of trace-accepting deterministic weak r-NDCMA (and even bounded NDCMA as in [12]), meaning this problem is undecidable.
- Representation of the trace language of HOSC[FOSC] with while loops using weak deterministic r-NDCMA.

We summarise the main closure and decidability results obtained below, alongside the closure results for the bounded NDCMA introduced in [14] for comparison.

	f-NDCMA	r-NDCMA	det. r-NDCMA	det. bdd. NDCMA
Complement	X	X	✓	✓
Intersection	✓	X	X	✓
Union	✓	✓	X	✓
Inclusion	X	X	X	✓
Equivalence	X	X	?	✓

Chapter 2

Well-quasi orders and WSTS

To lay the foundations for our study of automata over nested data, we first introduce the theory of well-quasi orders and well-structured transition systems. This framework will form the basis of our analysis of our new proposed models of NDCMA — in particular in proving decidability of emptiness of our models. For this purpose, we go through the relevant theory, covering the basics behind well-quasi orders and focusing on well-quasi orders on trees, and further introduce well-structured transition systems and the relevant decision problems for these systems. The results and definitions in this chapter were taken from [34], unless otherwise stated.

2.1 Well-quasi orders

We first introduce some equivalent definitions of well-quasi orders, and introduce basic machinery which will help us reason about these orders.

Definition 2.1.1 (Quasi order). A relation $\leq$ over a set A is a *quasi order* (qo) if and only if it is

(Q1) Reflexive: $x \leq x$ for all $x \in A$;

(Q2) Transitive: $x \leq y$ and $y \leq z$ implies $x \leq z$.

Definition 2.1.2 (Well-quasi ordering). A relation $\leq$ over a set A is a *well-quasi ordering* (wqo) if and only if

- It is a quasi order and
- For every infinite sequence $(x_i)_{i \in \mathbb{N}}$, there exists an increasing pair $i, j \in \mathbb{N}$, $i < j$ such that $x_i \leq x_j$.

Proposition 2.1.3. *Let $(A, \leq)$ be a qo. The following statements are equivalent:*

1. $(A, \leq)$ is a wqo.
2. Every infinite sequence $(x_i)_{i \in \mathbb{N}}$ over A contains an infinite increasing subsequence, that is $\exists i_0 < i_1 < i_2 < \dots$ such that $x_{i_0} \leq x_{i_1} \leq x_{i_2} \leq \dots$.
3. A is well founded, that is, there are no infinite strictly decreasing sequences $x_0 > x_1 > x_2 > \dots$ in A and A has no infinite antichains, that is there are no infinite sets $\{x_0, x_1, x_2, \dots\} \subseteq A$ of mutually incomparable elements, that is elements x_i, x_j such that it is neither the case that $x_i \leq x_j$ or $x_j \leq x_i$.

This characterisation can be used to immediately rule out an order being a well-quasi order. For example $(\mathbb{R}, \leq)$ and $(\mathbb{Z}, \leq)$ are clearly not well-quasi orders as they are not well founded. Similarly, $(\mathbb{N}, =)$ and $(\mathbb{N}, |)$, where $|$ is the divisibility relation, are clearly not wqos as there are infinitely many incomparable elements, the set of natural numbers $1, 2, 3, \dots$ for $(\mathbb{N}, =)$ and the set of primes $p_1, p_2, p_3, \dots$ for $(\mathbb{N}, |)$.

An example of a well-quasi order is $(\mathbb{N}, \leq)$ as this set is well-founded, and each element is comparable. In fact, it is obvious that every well order is a well-quasi order, as these are by definition well-founded and totally ordered, meaning every pair of elements is comparable. We are not just limited to wqos — a simple wqo which is not a well-order is the ordering $(A, =)$ for any finite set. We will further construct more examples of such wqos, by establishing ways to combine wqos.

Lemma 2.1.4. *If A is a finite set, then $(A, =)$ is a wqo.*

Proof. This is immediate from the fact that in an infinite sequence $(a_i)_{i \in \mathbb{N}}$, there will always be some repeated element since A is finite. $\square$

2.1.1 Combining well-quasi orders

We can further construct more complicated well-quasi orders from a given well-quasi order. We see this below for the subword and product orderings.

Definition 2.1.5. Let $(A, \leq)$ be a quasi order. The *subword ordering* $\leq_*$ on A^* is defined on sequences of A as

$$(a_1, \dots, a_n) \leq_* (b_1, \dots, b_m) \iff \exists i_1 < \dots < i_n, a_j \leq b_{i_j} \text{ for all } 1 \leq j \leq n.$$

Lemma 2.1.6 (Higman's Lemma). *If $(A, \leq)$ is a wqo, then so is $(A^*, \leq_*)$*

Definition 2.1.7. Let $(A, \leq_A)$, $(B, \leq_B)$ be quasi orders. The *product ordering* $\leq_{A \times B}$ on $A \times B$ is defined as

$$(a_1, b_1) \leq_{A \times B} (a_2, b_2) \iff a_1 \leq_A a_2 \text{ and } b_1 \leq_B b_2.$$

Lemma 2.1.8 (Dickson's Lemma). *If $(A, \leq_A)$, $(B, \leq_B)$ are wqos, then so is $(A \times B, \leq_{A \times B})$*

2.1.2 Well-quasi orders on trees

We further study well-quasi orderings on trees. For the bounded NDCMA proposed in [14], the proof of emptiness relied on using a wqo on bounded trees. We will investigate wqos on trees, investigating the orderings on bounded trees given in [14, 25], which we denote $\leq_C, \leq_M$. We detail one possible way to prove that $\leq_C, \leq_M$ are indeed wqos, as this fact was claimed but not proven in the respective papers. We further show that these are not wqos on unbounded trees. Furthermore, Kruskal's tree ordering $\leq_T$ is introduced, which is a wqo over all finite trees.

We work with finite rooted trees labelled from a fixed set A . There are two ways that we can represent trees. The typical (graph-theoretic) way is to define a tree as a tuple $T = \langle D, \text{pred} \rangle$, where D is a finite set with a distinguished root r and $\text{pred} : D \rightarrow D$ is defined on all D except for the root r , such that for all d there exists some $n \in \mathbb{N}$, such that $\text{pred}^n(d) = r$. We refer to such a node as a level n node, and say that r is a level 0 node. Note that we can distinguish $r \in D$ by the fact that $\text{pred}(r)$ is undefined. Then we can further consider trees labelled by a set A by assigning a function $f : D \rightarrow A$.

Alternatively, we can define labelled trees recursively using the definition

$$T := a \mid (a, (T_0, \dots, T_n))$$

where $a \in A$, $T_0, \dots, T_n$ are trees and $n \in \mathbb{N}$. This alternate definition is used in [25], and in this case the children of a node are also equipped with an implicit ordering $T_0, \dots, T_n$.

There is a clear way to go from one definition to another. Note that when converting from the graph theoretic representation to a tuple, we need to give each node's children some arbitrary ordering. For the same reason, when converting the other way we lose the 'ordering' information given. We denote the set of all finite trees as $\mathcal{T}$, where the type of representation being used can be inferred from the context. We will further use $\leq_{M,*}$ to denote the subword ordering over trees with the underlying order $\leq_M$, and use similar notation throughout this section for different constructions and orders.

Definition 2.1.9 (Rooted Tree Embedding Ordering [25]). The relation $\leq_M$ on trees labelled by A maps the root to the root, leaves to leaves and preserves the levels of the nodes. Formally, this is defined on ordered trees as follows:

- (M1) $a \leq b$ implies $a \leq_M b$,
- (M2) $a \leq b$ and $(T_1, \dots, T_n) \leq_{M,*} (T'_1, \dots, T'_m)$ for $n \geq 1$ implies $(a, (T_1, \dots, T_n)) \leq_M (b, (T'_1, \dots, T'_m))$.

Definition 2.1.10. The *depth* of a tree $T \in \mathcal{T}$ is the length of the longest path from the root to a leaf node. We can also define this recursively on the tuple representation as $\text{depth}(a) = 1$ and $\text{depth}((a, (T_0, \dots, T_n))) = 1 + \max(\text{depth}(T_0), \dots, \text{depth}(T_n))$. We further refer to the set of all trees with depth bounded by n as $\mathcal{T}_n$.

Lemma 2.1.11. *If $(A, \leq)$ is a wqo, $\leq_M$ is a wqo on the set of trees $\mathcal{T}_n$ with depth bounded by n .*

Proof. This is claimed in [25] but the proof is omitted. We prove this claim using induction. For $n = 1$, it is clear $\leq_M$ reduces to the wqo on A hence is clearly a

wqo. Suppose $(\mathcal{T}_k, \leq_M)$ forms a wqo, and consider all the labelled trees of depth bounded by $k + 1$.

Now consider a sequence $(T_i)_{i \in \mathbb{N}}$. If there are infinitely many trees of depth bounded by k we are done as since $(\mathcal{T}_k, \leq_M)$ forms a wqo, there exists some T_i, T_j with $i < j$ and $T_i \leq_M T_j$. Otherwise, there exists a subsequence of trees $(T_{k_i})_{i \in \mathbb{N}}$ of depth $k + 1$. These trees can be written as $(a, (T_1, \dots, T_n))$, with $T_i \in \mathcal{T}_k$, noting $n \geq 1$ as our trees have depth $k + 1$.

By definition, we have $(a, (T_1, \dots, T_n)) \leq_M (b, (T'_1, \dots, T'_m))$ if and only if $a \leq b$ and $(T_1, \dots, T_n) \leq_{M,*} (T'_1, \dots, T'_m)$. Now, by Higman's Lemma 2.1.6, $\leq_{M,*}$ is a wqo on $\mathcal{T}_k^*$ since $\leq_M$ is a wqo on $\mathcal{T}_k$ by the inductive hypothesis. Therefore by Dickson's Lemma 2.1.8, $\leq_M$ is a well ordering on our subsequence, so there exists $T_{k_i} \leq T_{k_j}$ with $i < j$, as required. □

We present a different ordering on trees now, which does not make use of the ordered tuple definition. This is the ordering utilised in the proof of decidability of emptiness of the bounded NDCMA from [14].

Definition 2.1.12 (Tree Homomorphism Ordering). Let T_1, T_2 be trees, and $(A, \leq)$ a quasi order, along with labelling functions $f_i : T_i \rightarrow A$ for $i \in \{1, 2\}$. We let $T_1 \leq_C T_2$ whenever there exists an injective tree homomorphism $\phi : T_1 \rightarrow T_2$, such that for all s , $f_1(s) \leq f_2(\phi(s))$. That is, there exists injective ϕ which maps the root of T_1 to the root of T_2 , and preserves the predecessor relation on nodes, $\phi(\text{pred}(s)) = \text{pred}(\phi(s))$.

We would like to compare $\leq_C$ with $\leq_M$, but to do so we need to find an analogous definition for the tuple representation of trees. To do this, we define relation similar to the subword ordering $\leq_*$ on sequences, which does not enforce that the order of elements in the sequence is the same, but still enforces that two distinct subtrees are mapped to distinct subtrees.

Definition 2.1.13. Let $(A, \leq)$ be a quasi order. The *multiset subword ordering* $\leq_\sim$ on A^* is defined on sequences of A as

$$(a_1, \dots, a_n) \leq_\sim (b_1, \dots, b_m) \iff \exists i_1, i_2, \dots, i_n, \ i_j \neq i_k \text{ for all } j \neq k \text{ and} \\ a_j \leq b_{i_j} \text{ for all } 1 \leq j \leq n.$$

Alternatively, if $\text{mSet}(a_1, \dots, a_n)$ denotes the multiset obtained from the sequence, then $(a_1, \dots, a_n) \leq_{\sim} (b_1, \dots, b_m)$ if and only if $\text{mSet}(a_1, \dots, a_n) \subseteq \text{mSet}(b_1, \dots, b_m)$, where $\subseteq$ denotes the subset relation on multisets. This is studied in [26].

Definition 2.1.14 (Alt Tree Homomorphism Ordering). We define the ordering $\leq_C$ on ordered trees as follows:

- (C1) $a \leq b$ implies $a \leq_C b$ and $a \leq_C (b, (T'_1, \dots, T'_m))$
- (C2) $a \leq b$ and $(T_1, \dots, T_n) \leq_{C, \sim} (T'_1, \dots, T'_m)$ for $n \geq 1$ implies $(a, (T_1, \dots, T_n)) \leq_C (b, (T'_1, \dots, T'_m))$

The extra condition in (C1) can be seen as relaxing the condition in (C2) to include $n = 0$, representing $a = (a, \epsilon)$. This essentially allows us to add leaves to a graph, while still preserving the order.

It follows from the definition that $T_1 \leq_C T_2$ for any tuple representations of T_1, T_2 if and only if $T_1 \leq_C T_2$ in the graph presentation. However this definition makes it clearer to see that $\leq_C$ is a refinement of $\leq_M$, which allows us to conclude that $\leq_C$ is a wqo on bounded trees.

Lemma 2.1.15. *Suppose $(A, \leq)$ is a wqo. Then, $(\mathcal{T}_n, \leq_C)$ forms a wqo.*

Proof. We first claim that it is easy to see that $\leq_C$ is well-founded. If we have $T_1 \leq_C T_2$, then there is an injection ϕ from T_1 to T_2 , so $|T_1| \leq |T_2|$, that is T_2 has at least as many nodes as T_1 . Now, if $|T_1| = |T_2|$, then ϕ is a bijection, hence its inverse ϕ^{-1} is also a tree homomorphism. Suppose we have an infinitely decreasing sequence $T_0 > T_1 > T_2 > T_3 > \dots$. We know $|T_0| \geq |T_1| \geq |T_2| \geq \dots$. Since the natural numbers are well-founded, there exists some k such that $|T_k| = |T_{k+1}| = |T_{k+2}| = \dots$, but this would imply that there exists some infinitely decreasing sequence in A , contradicting that $(A, \leq)$ is a wqo.

To see that $\leq_C$ has no infinite antichains, it is enough to note that $T_1 \leq_M T_2$ implies $T_1 \leq_C T_2$. This is clear from the definition, along with the fact that $s \leq_{C, *} s'$ implies $s \leq_{C, \sim} s'$. Now, suppose there exists an infinite set $S = \{T_0, T_1, T_2, \dots\}$ such that for all $i \neq j$, it is not the case that $T_i \leq_C T_j$ or $T_j \leq_C T_i$. Then it must be the case that $T_i \leq_M T_j$ and $T_j \leq_M T_i$ do *not* hold for any arbitrary orderings imposed on T_i, T_j , so S is an infinite antichain for $\leq_M$, contradicting the fact that $(\mathcal{T}_n, \leq_M)$ is a wqo. $\square$

Neither $\leq_C$ or $\leq_M$ are wqo on unbounded trees, even labelled on the singleton $A = \{a\}$. These are both clearly well-founded, so the issue arises in the condition that they have no infinite antichains.

Example 2.1.16. For $\leq_M$ it is clear that the set of paths $\{P_1, P_2, P_3, \dots\}$, where $P_1 = a$ and $P_{i+1} = (a, (P_i))$ form an antichain, as for $T_1 \leq_M T_2$ to hold, the root must be mapped to the root and the leaves must be mapped to leaves. Clearly there is no such mapping for P_i, P_j with $i \neq j$.

To see that $\leq_C$ is not a wqo, we can similarly construct a set of infinite graphs unrelated to each other with $\leq_C$ as in Figure 2.1. This example is adapted from [35], answering the same question for the induced subgraph ordering, which is clearly a refinement of $\leq_C$.

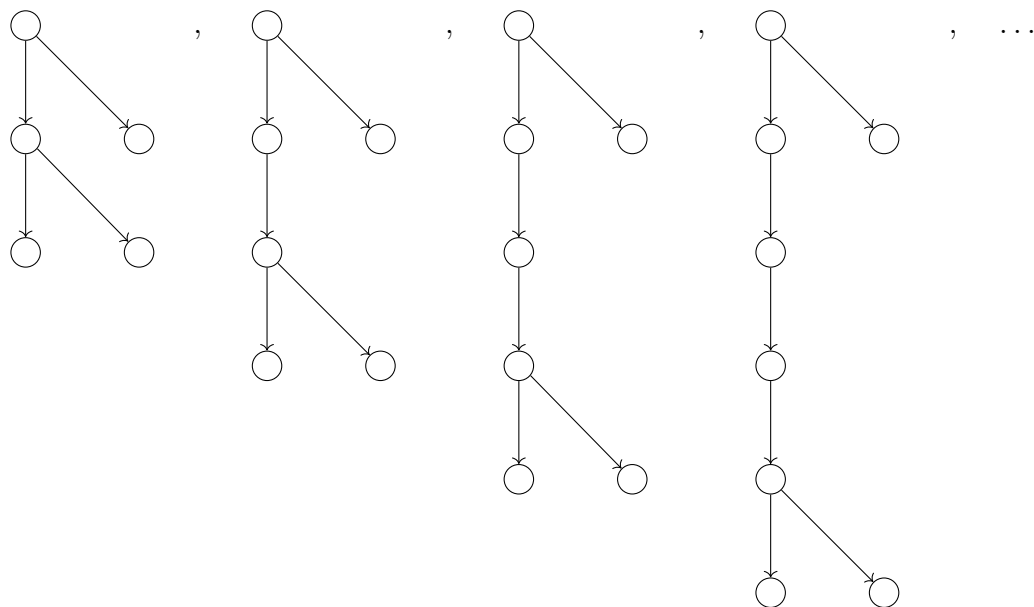


Figure 2.1: Infinite antichain for $\leq_C$

These results make $\leq_C$ and $\leq_M$ unsuitable for comparing trees of unbounded depth. However it is a well known result from Kruskal that the set of finite labelled trees ordered by *homeomorphic embeddings* ($\leq_T$) is a wqo. This ordering can be further seen as a refinement of $\leq_C$ as $T_1 \leq_C T_2$ implies $T_1 \leq_T T_2$.

Definition 2.1.17 (Homeomorphic Embedding Ordering). Given two trees T_1, T_2 , we say that ϕ is a homeomorphic embedding from T_1 into T_2 if ϕ is injective and

preserves the tree ordering on the vertices of T_1 . That is, if $\text{pred}^n(u) = v$, then there exists some m such that $\text{pred}^m(\phi(u)) = \phi(v)$. Moreover, given a set $(A, \leq)$ and labelling functions $f_i : T_i \rightarrow A$, we say that $T_1 \leq_T T_2$ if there exists a homeomorphic embedding $\phi : T_1 \rightarrow T_2$ such that $f_1(v) \leq f_2(\phi(v))$ for all v .

Alternatively, we can give an inductive definition as before.

- (C1) $a \leq b$ implies $a \leq_T b$ and $a \leq_T (b, (T'_1, \dots, T'_m))$
- (C2) $a \leq b$ and $(T_1, \dots, T_n) \leq_{T, \sim} (T'_1, \dots, T'_m)$ for $n \geq 1$ implies $(a, (T_1, \dots, T_n)) \leq_T (b, (T'_1, \dots, T'_m))$
- (C3) $T \leq_T T_i$ implies $T \leq_T (b, (T_1, \dots, T_i, \dots, T_n))$

This definition makes it clear that $\leq_T$ is a refinement of $\leq_C$. The difference between the two is condition (C3) which essentially allows for a tree to be embedded in another without maintaining the strict pred relationships.

Theorem 2.1.18 (Kruskal's Tree Theorem [24]). *If $(A, \leq)$ forms a wqo, then so does $(\mathcal{T}, \leq_T)$, over all finite labelled trees.*

Remark 2.1.19. There are many ways to present Kruskal's theorem. One of the most popular (and simplest) proofs of this theorem [27] was done using the finer order $\leq_{T,*}$ in the definition instead of the multiset based one, $\leq_{T,\sim}$. Of course, this gives a different ordering, but it is easy to show that the ordering presented above has no infinite antichains if the alternate ordering has no infinite antichains by the same techniques as Lemma 2.1.15. In fact this is just an application of the fact that for $\leq, \preceq$ defined on the same sets, if $a \leq b$ implies $a \preceq b$ then $\preceq$ having infinite antichains implies $\leq$ does too.

The fact that $\leq_T$ is well-founded is easy to prove, noting that in an infinite descending sequence, the depth of a tree always has to eventually decrease. Hence well-foundedness follows by well-foundedness of the natural numbers.

2.2 Well-Structured Transition Systems

Definition 2.2.1. A *transition system* $\langle S, \rightarrow \rangle$ consists of a set S and a binary relation $\rightarrow \subseteq S \times S$. When further equipped with a wqo $\leq$ on S we call the

triple $\mathcal{S} = \langle S, \rightarrow, \leq \rangle$ a *well-structured transition system (WSTS)* if it satisfies the following (upwards) compatibility condition:

$$s \rightarrow s' \text{ and } s \leq t \text{ implies } \exists t' \geq s', t \rightarrow t'.$$

Remark 2.2.2. We remark that there are various different definitions of well-structured transition systems. We take the definition from [34], and note that this is different to the one taken in [17]. In this paper, well-structured transition systems are restricted to be image-finite, meaning $s \rightarrow t$ for only finitely many t and moreover have a weaker compatibility condition. In the original paper and thesis introducing bounded NDCMA [14, 12] this definition is used, however the WSTS described in this paper still satisfies our definition. Note that the WSTS described in Chapter 3 will not in fact be image-finite.

Definition 2.2.3. Given a set $U \subseteq A$ and a $\text{qo } \leq$ on A , the *upward-closure* of U , denoted $\uparrow U$ is the set

$$\uparrow U = \{x' \in A \mid \exists x \in U, x \leq x'\}.$$

Whenever $U = \uparrow U$, we say that U is *upward-closed*. Moreover, if $U = \uparrow S$, we say S is a *basis* of U .

2.2.1 Decision problems on WSTS

We present two common decision problems for WSTS — namely those of *termination* and *coverability*. These are widely studied as certain conditions on the WSTS makes them decidable.

Definition 2.2.4 (Termination). Given a WSTS, $\mathcal{S} = \langle S, \rightarrow, \leq \rangle$ and a state s_0 , the *termination problem* is defined as follows: Is every possible transition sequence $s_0 \rightarrow s_1 \rightarrow \dots$ finite?

Definition 2.2.5 (Coverability). Given a WSTS, $\mathcal{S} = \langle S, \rightarrow, \leq \rangle$ and two states $s, t \in S$, the *coverability problem* is defined as follows: Does there exist a transition sequence $s \rightarrow^* s'$ such that $s' \geq t$?

We further look at some properties of well-structured transition systems that are used to ensure decidability of these problems. These depend on the successor and predecessor sets of a state/set of states, as being able to reason about these sets effectively is the key to the decidability results of the above problems.

Definition 2.2.6. Let $\mathcal{S} = \langle S, \rightarrow, \leq \rangle$ be a WSTS. For a state $s \in S$, we define its successor set $\text{Post}(s)$ to be the set

$$\text{Post}(s) = \{s' \in S \mid s \rightarrow s'\}.$$

Moreover, for a set of states $U \subseteq S$, we define its predecessor set $\text{Pre}(U)$ to be the set

$$\text{Pre}(U) = \{s' \in S \mid \exists s \in U, s' \rightarrow s\}.$$

Definition 2.2.7. $\mathcal{S} = \langle S, \rightarrow, \leq \rangle$ has an *effective pred-basis* if there exists an algorithm accepting any state $s \in S$ and returning $\text{pb}(s)$, a finite basis of $\uparrow \text{Pre}(\uparrow\{s\})$.

Remark 2.2.8. By Lemma 1.16 in [34], we have that $\uparrow \text{Pre}(\uparrow\{s\}) = \text{Pre}(\uparrow\{s\})$, so to find $\text{pb}(s)$, we only need to find a finite basis for $\text{Pre}(\uparrow\{s\})$.

Definition 2.2.9. $\mathcal{S} = \langle S, \rightarrow, \leq \rangle$ is *image finite* if $\text{Post}(s)$ is finite for all $s \in S$. Furthermore, we say that $\mathcal{S}$ is *post-effective* if $\text{Post}(s)$ can be effectively computed from s .

Theorem 2.2.10 (Decidability of Termination). *If $\mathcal{S} = \langle S, \rightarrow, \leq \rangle$ is an image-finite, post-effective WSTS with decidable $\leq$, then termination in $\mathcal{S}$ is decidable.*

Theorem 2.2.11 (Decidability of Coverability). *If $\mathcal{S} = \langle S, \rightarrow, \leq \rangle$ is a WSTS with an effective pred-basis and decidable $\leq$, then coverability in $\mathcal{S}$ is decidable.*

Remark 2.2.12. Techniques on WSTS have been developed to obtain complexity bounds of the above decision problems by limiting the length of *bad sequences* of decreasing/incomparable elements [32]. These can be applied for WSTS using wqos on bounded trees using techniques from [33], or applying directly the upper bound from Section 4, [19]. Although we do not detail this process here, it is clear that these techniques would be useful for finding upper bounds on the complexity of deciding non-emptiness for the NDCMA models in [14] and in Chapter 4.

Chapter 3

Flexible NDCMA

In this chapter we detail our first attempt in extending the NDCMA from [12] to allow for parsing trees of arbitrary depth. From now on, we will refer to these automata as *bounded NDCMA*, and refer to our new variants of NDCMA as *unbounded data NDCMA*. Another difference in our new models is that while the automata in [14] store the last seen state in the class memory, we will follow [9] and store labels from a finite set $\mathcal{M}$. This allows for more expressivity while still retaining the core idea.

The automata in this chapter will be called *flexible NDCMA*, as in this model a data value will be able to access the labels of all its ancestors to determine its transition. Moreover, a node will not be able to distinguish between its different ancestors. We will prove decidability of emptiness of these automata, explore closure results and finally detail why these automata are not ideal for our proposed application of unbounded NDCMA.

3.1 Introducing f-NDCMA

First we formally define the data set $\mathcal{D}$ which we will be using throughout Chapters 3 and 4. This data set will also specify the underlying hierarchical structure, which is a forest structure. Our forest is made up of infinitely many infinitely full, which corresponds to us always being able to obtain a fresh data value at any position.

Definition 3.1.1. We first define a *nested data set* $\mathcal{D} = \langle D, \text{pred} \rangle$ to be a forest of infinitely many trees, each of which are *full*, that is each node has infinitely many

children. We typically refer to root nodes as level 0 nodes, their children as level 1 nodes and so on.

Given a set S , we will take $S_\perp$ to denote the set $S \cup \{\perp\}$ where $\perp$ is not an element of S .

Definition 3.1.2. Let $k \geq 1$ be a fixed natural number. A *k-flexible Nested Data Class Memory Automaton (f-NDCMA)* over the alphabet $\Sigma \times \mathcal{D}$, where $\mathcal{D}$ is a nested data set is a tuple $\mathcal{A} = \langle Q, \Sigma, \mathcal{M}, \delta, q_0, \mathcal{M}_L, F_G \rangle$ where Q is a finite set of states, $\mathcal{M}$ is a finite set of labels, q_0 is the initial state, $F_G \subseteq Q$ consists of the globally accepting states and $\mathcal{M}_L \subseteq \mathcal{M}$ consists of the locally accepting labels. δ is the transition function, which is the union $\delta = \bigcup_{i=1}^k \delta_i$ and

$$\delta_i : Q \times \Sigma \times (\mathcal{M}_\perp)^i \rightarrow \mathcal{P}(Q \times (\mathcal{M}_\perp)^i).$$

Similar to CMA, we say $\mathcal{A}$ is *deterministic* if each set in the image of δ is a singleton, and $\mathcal{A}$ is *weak* if all labels are locally accepting, that is $\mathcal{M}_L = \mathcal{M}$. We use $q \xrightarrow[(u_1, \dots, u_i)]{(v_1, \dots, v_i)} p$ to denote $(p, (u_1, \dots, u_i)) \in \delta_i(q, (v_1, \dots, v_i))$.

Definition 3.1.3. Given a flexible NDCMA $\mathcal{A} = \langle Q, \Sigma, \mathcal{M}, \delta, q_0, \mathcal{M}_L, F_G \rangle$, a *configuration* is a pair (q, f) , where $q \in Q$ and $f : D \rightarrow \mathcal{M}_\perp$ is a *class memory function*, that is $f(d) \neq \perp$ for only finitely many d . The initial configuration is (q_0, f_0) , where $f_0(d) = \perp$ for all $d \in D$. A configuration (q, f) is *accepting* if $q \in F_G$ and $f(D) \subseteq \mathcal{M}_L \cup \{\perp\}$.

The automaton can transition from configuration (q, f) to (q', f') on reading input (a, d) where $a \in \Sigma$, $d \in D$ if

- $q \xrightarrow[u_1, \dots, u_n]{a, (v_1, \dots, v_n)} q'$, that is $(q', a, (u_1, \dots, u_n)) \in \delta_n(q, (v_1, \dots, v_n))$,
- there exist integers $0 = m_1 < m_2 < \dots < m_n$, such that $f(\text{pred}^{m_i}(d)) = v_i$ and
- $f' = f[\text{pred}^{m_1}(d) \mapsto u_1, \dots, \text{pred}^{m_n}(d) \mapsto u_n]$.

Intuitively, we can transition from a data value d if the path $d = d_0, d_1, \dots, d_k = r$ from d to the root contains a subsequence $d_{m_1}, d_{m_2}, \dots, d_{m_n}$ which has the expected

state labels. The restriction that $m_1 = 0$ ensures the first value v_1 is matched necessarily to the value of d .

A *run* $(q_0, f_0), (q_1, f_1), \dots, (q_n, f_n)$ is a sequence of configurations such that (q_0, f_0) is the initial configuration and (q_{i+1}, f_{i+1}) is a valid next configuration from (q_i, f_i) . Moreover, this run is *accepting* if and only if (q_n, f_n) is an accepting configuration. For $w \in (\Sigma \times \mathcal{D})^*$, we say that $\mathcal{A}$ accepts w if and only if there exists an accepting run in which the input at each step i is $w[i]$. Moreover, we let $\mathcal{L}(\mathcal{A}) \subseteq (\Sigma \times \mathcal{D})^*$ be the set of words accepted by $\mathcal{A}$.

Example 3.1.4. We exhibit the power of f-NDCMA using a simple example with two states, illustrated in Figure 3.1. In figures, we will denote a transition $q \xrightarrow[s]{a, t} p$ by marking the arc with ‘ $a, t/s$ ’, where t is the tuple of labels to be seen and s is the relabelling. Let $\mathcal{A}_1 = \langle \{q_0, q_1\}, \{0\}, \mathcal{M}, \delta_1, q_0, \mathcal{M}_L, \{q_0\} \rangle$ with $\mathcal{M} = \mathcal{M}_L = \{next, seen\}$, and only two transitions in δ ,

$$q_0 \xrightarrow[(seen, next)]{0, (\perp, \perp)} q_1, \quad q_1 \xrightarrow[(seen)]{0, (next)} q_0.$$

We will illustrate some simple runs of $\mathcal{A}_1$, using the data values $d_1, d_2, d_3, d_4, d_5 \in D$ such that $\text{pred}(d_{i+1}) = d_i$ for $1 \leq i < 5$. Note that since our alphabet Σ is a singleton we will omit the alphabet in our analysis as this does not change the resulting language. Initially, the class memory function f is set to $\perp$ for all data values, so $f(d_i) = \perp$ for each i . Moreover, we begin in state q_0 , as this is the start state of the automaton.

Suppose we see the data value d_4 initially. Since $f(d_4) = \perp$, the first value matches the first term of the tuple $(\perp, \perp)$. To use the transition we need to pick some ancestor of d_4 with the value $\perp$. Hence we have the free choice from d_1, d_2

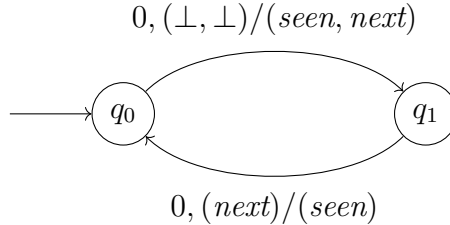


Figure 3.1: Example of a 2-flexible NDCMA, $\mathcal{A}_1$

and d_3 . We cannot match with d_5 , as it is not an ancestor of d_4 . Suppose for this run we match with d_4, d_2 , noting $(f(d_4), f(d_2)) = (\perp, \perp)$ as required. We transition to state q_1 , setting $f(d_4) = \text{seen}$ and $f(d_2) = \text{next}$. Then, in the next transition we have to see the data value d_2 , as it is the only value with label next , and change its label to seen in moving to q_0 . Now we have

$$f(d_i) = \begin{cases} \text{seen} & \text{if } i \in \{2, 4\}; \\ \perp & \text{otherwise.} \end{cases}$$

Suppose next we see data value d_5 . Now we can't match with the ancestor d_2 or d_4 , as $(f(d_5), f(d_2)) = (f(d_5), f(d_4)) = (\perp, \text{seen}) \neq (\perp, \perp)$, but we can still match with d_3 or d_1 . If we match with d_1 , then we set $f(d_5) = \text{seen}$, $f(d_1) = \text{next}$ and move to state q_1 . Similar to before, we can now only see d_1 as input, transitioning to q_0 . Since q_0 is an accepting state, we have described an accepting run $(q_0, f_0), (q_1, f_0[d_4 \mapsto \text{seen}, d_2 \mapsto \text{next}]), (q_0, f_0[d_i \mapsto \text{seen} \text{ for } i \in \{2, 4\}]), (q_1, f_0[d_i \mapsto \text{seen} \text{ for } i \in \{2, 4, 5\}, d_1 \mapsto \text{next}]), (q_1, f_0[d_i \mapsto \text{seen} \text{ for } i \in \{1, 2, 4, 5\}])$ given the word $d_4 d_2 d_5 d_1$.

Extrapolating this analysis, it can easily be seen that the language accepted by $\mathcal{A}_1$ (omitting the singleton alphabet term) is the language consisting of new data values such that every odd value is followed by the data value of one of its ancestors, that is,

$$\mathcal{L}(\mathcal{A}_1) = \{d_1, \dots, d_{2k} \mid k \in \mathbb{N}, d_i \neq d_j \text{ for } i \neq j \text{ and} \\ \text{for all } 1 \leq i < k, d_{2i} \text{ is an ancestor of } d_{2i-1}\}.$$

Given this, it is clear that $d_5 d_4 d_3 d_1$, $d_3 d_1 d_4 d_2$ are accepted words but $d_4 d_5$ and $d_3 d_1 d_2 d_4$ are not. It is also clear that these automata can express different words to the bounded NDCMA from [14], as $\mathcal{L}(\mathcal{A}_1)$ can express data words with values of an unbounded depth, such as dd' where $\text{pred}^n(d) = d'$.

A feature of f-NDCMA that does not emerge in 2-flexible NDCMA is that the order of the ancestors matters. To see this, consider the 3-flexible NDCMA $\mathcal{A}_2 = \langle \{q_0, q_1, q_2\}, \{0\}, \mathcal{M}, \delta_2, q_0, \mathcal{M}_L, \{q_0\} \rangle$ with $\mathcal{M} = \mathcal{M}_L = \{\text{next}, \text{first}, \text{second}\}$, with the transitions as in Figure 3.2. We can see the effects of order here as when we transition from q_0 to q_1 given some value d , the ancestors of d that we match with, c, c' , must be such that c' is an ancestor of c , and are given labels

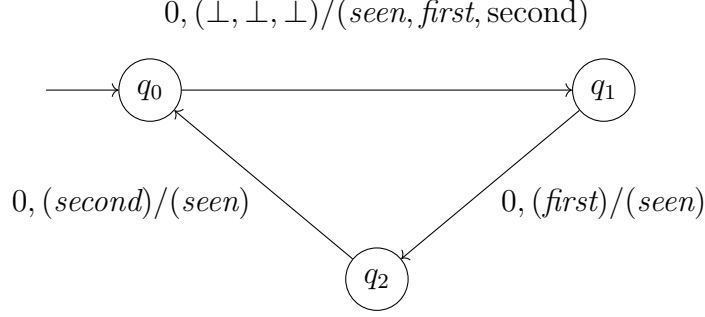


Figure 3.2: Example of a 3-flexible NDCMA $\mathcal{A}_2$

$(f(c), f(c')) = (first, second)$ respectively. Order matters in this automaton, and in fact the language accepted by $\mathcal{A}_2$ is

$$\begin{aligned} \mathcal{L}(\mathcal{A}_2) = \{d_1, \dots, d_{3k} \mid k \in \mathbb{N}, d_i \neq d_j \text{ for } i \neq j \text{ and} \\ \text{for all } 1 \leq i < k, d_{3i} \text{ is an ancestor of } d_{3i-1} \text{ and} \\ d_{3i-1} \text{ is an ancestor of } d_{3i-2}\}. \end{aligned}$$

Hence if we have $d_1, d_2, d_3, d_4, d_5 \in D$ as before, $d_5 d_4 d_2$ is an accepted word, but $d_5 d_2 d_4$ is not.

In both automata, we can see inherent non-deterministic behaviour in the fact that we are free to choose any ancestor(s) of our data value d to mark. We will highlight this further later on, as this behaviour will lead us to consider different automata models.

3.2 WSTS for f-NDCMA

To prove decidability of non-emptiness of an f-NDCMA, we reduce this problem to a covering problem of a WSTS. To do this, we work similarly to [12, 14], and represent the class memory function $f : D \rightarrow \mathcal{M}_\perp$ with a labelled tree. We similarly identify the root node with some unique symbol $\# \notin Q$, so that we have a tree instead of a forest.

In our case, it is possible that $f(d) \neq \perp$ and $f(\text{pred}(d)) = \perp$, however for each class memory function f and every path, we will eventually reach a node such that all its successors are labelled with $\perp$. Hence for each f we can construct a finite

graph T_f labelled by $\mathcal{M}_\perp \cup \{\#\}$ such that the leaves are all labelled by $\mathcal{M}$ (or $\#$ in the case where the graph consists only of a root). Let $\mathcal{TL}$ be the set of graphs labelled by $\mathcal{M}_\perp$ except for the root which is labelled $\#$, such that the leaves are not labelled by $\perp$. Given this set, we construct a WSTS where the states represent configurations, and the transitions correspond to automaton transitions.

Lemma 3.2.1. *Given an f -NDCMA $\mathcal{A}$, $\mathcal{S}_\mathcal{A} = \langle S, \rightarrow, \leq \rangle$ as defined below is a WSTS.*

- $S = Q \times \mathcal{TL}$,
- $(q, T) \rightarrow (q', T')$ if and only if there exist class memory functions f, f' where $T_f = T$ and $T_{f'} = T'$, such that there is a transition from configuration (q, f_T) to $(q', f_{T'})$.
- $(q, T) \leq (q', T')$ if and only if $q = q'$ and $T \leq_T T'$ as in Definition 2.1.17, where $(A, \leq) = (Q_\perp \cup \{\#\}, =)$.

Proof. We see that $\leq$ is a wqo on S , first by noting $(\mathcal{M}_\perp \cup \{\#\}, =)$ is a wqo since $\mathcal{M}_\perp \cup \{\#\}$ is a finite set, so we can apply Kruskal's Tree Theorem 2.1.18 to conclude that $\leq_T$ is a wqo over all trees labelled with $\mathcal{M}_\perp \cup \{\#\}$. Now note that for any wqo $(B, \leq)$, if $U \subseteq B$, then $(U, \leq)$ is also a wqo, so in fact $\leq_T$ is a wqo over $\mathcal{TL}$. It is now only left to remark that $(Q, =)$ is a wqo as Q is finite, and $\leq$ is the product ordering of $(Q, =)$ and $(\mathcal{TL}, \leq_T)$, hence by Dickson's Lemma 2.1.8, $\leq$ is a wqo.

Furthermore, we need to prove the upward compatibility condition, which states that $(q, T) \rightarrow (q', T')$ and $(q, T) \leq (p, R)$ implies there exists some $(p', R') \geq (q', T')$ such that $(p, R) \rightarrow (p', R')$. Now, $(q, T) \rightarrow (q', T')$ means that there exist $a \in \Sigma$, $v_1, \dots, v_n \in \mathcal{M}_\perp$ with $q \xrightarrow[(u_1, \dots, u_n)]{a, (v_1, \dots, v_n)} q'$ and a node d in T , integers $0 = m_1 < m_2 < \dots < m_n$ such that the vertices $(d_1, \dots, d_n) = (\text{pred}^{m_1}(d), \dots, \text{pred}^{m_n}(d))$ are labelled with $v_1, \dots, v_n$. An exception to this is if the data value d is 'out of scope' of T , which means that there is some $k \geq 1$ such that $(v_1, \dots, v_n) = (\perp, \dots, \perp, v_{k+1}, \dots, v_n)$. In this case, $d_1, \dots, d_k$ are vertices of the tree T extended by a path of $\perp$ s.

Moreover, since T' is a tree representing the successor configuration of the class memory function represented by T , T' must be the same tree as T , except the nodes $d_1, \dots, d_n$ are labelled with $u_1, \dots, u_n$. If T had to be extended by $\perp$ s, note that we truncate the new path at d_1 , as we have the condition that the leaves of the tree are always labelled by $\mathcal{M}$.

Now, $(q, T) \leq (p, R)$ means that $q = p$ and $T \leq_T R$. The second condition assures us that the vertices $d_1, \dots, d_n$ in T with labels $v_1, \dots, v_n$ are mapped by a homeomorphic embedding $\phi : T \rightarrow R$ to vertices $d'_1, \dots, d'_n$ with the same labels in R . Note that if we need to extend T to include vertices with labels $\perp$, we extend R in the same way. Then, we can apply the same transition to R as was applied to T , on the data value $\phi(d)$, to obtain R' , which has the same vertices of R , with $d'_1, \dots, d'_n$ relabelled $u_1, \dots, u_n$. We have that $(p, R) = (q, R) \rightarrow (q', R')$.

Finally, we need to argue that $T' \leq_T R'$ to obtain that $(q', R') \geq (q', T')$. It is straightforward to construct a homeomorphic embedding $\phi' : T' \rightarrow R'$ from the homeomorphic embedding $\phi : T \rightarrow R$, by mapping the untouched vertices $d' \in T'$ in the same exact way, mapping d_i in T' to d'_i in R' and any additional ‘added’ $\perp$ vertices to the corresponding added $\perp$ vertices. Note that this last part works since we extended R in the exact same way as T . Hence we have upward compatibility, so $\mathcal{S}_{\mathcal{A}}$ is a WSTS. $\square$

Theorem 3.2.2. *The covering problem for $\mathcal{S}_{\mathcal{A}}$ is decidable.*

Proof. By Theorem 2.2.11, to prove that coverability is decidable, it is enough to show that $\mathcal{S}$ has an effective pred-basis and decidable $\leq$. The fact that $\leq$ is decidable is immediate as $\leq_T$ is decidable on trees, as given T_1, T_2 we can decide if $T_1 \leq_T T_2$ by iterating through each injection from T_1 to T_2 and checking whether it is a homeomorphic embedding.

Now, let $s = (p, T)$ be fixed. We establish an algorithm for calculating $\text{pb}(s)$ such that $\uparrow \text{pb}(s) = \uparrow \text{Pre}(\uparrow \{s\}) = \text{Pre}(\uparrow \{s\})$. We do this by finding a basis $\text{pb}_{\delta_n}(s)$ for each n and letting $\text{pb}(s) = \bigcup_{1 \leq n \leq k} \text{pb}_{\delta_n}(s)$. First note if p is not in the range of δ_n , we put $\text{pb}_{\delta_n}(s) = \emptyset$, as each $s' = (p', T')$ such that $s \leq s'$ has $p' = p$ hence is unreachable using the transition δ .

Consider each the transition $q \xrightarrow{(u_1, \dots, u_n)}^{a, (v_1, \dots, v_n)} p$ in δ_n . We will construct the possible previous tree configurations R of this transition given the resulting tree T . Now,

consider the set of all paths $\mathcal{P}$ starting from a leaf node of T , ending at the root node. This set is finite, as T is a finite tree. Fix a particular path $P \in \mathcal{P}$. We would like to identify all possible ways we could have transitioned from (q, R) to (p, T') for $T \leq_T T'$, using vertices in the path P . Every path in T is mapped with the homeomorphic embedding ϕ to a subpath of a path from leaf to root in T' , which we will refer to as the corresponding path.

We account for the fact that the transition made can reflect on the vertices of T' which are not in T . Our path $P = d_1, d_2, \dots, d_m = r$ will have some number of subpaths $d_{j_1}, \dots, d_{j_k}$ with the labels $u_{j_1}, \dots, u_{j_k}$, $j_1 < j_2 < \dots < j_k$. We iterate through all these possible subpaths, for all possible $0 \leq k \leq n$. This k represents the number of corresponding nodes of T which were modified in the transition to T' .

Now, we must consider all possible ways to add $n - k$ new vertices to the path. These will be labelled with labels u_j according to the position they are created in. Each possible configuration gives us a tree H with identified vertices $d_1, \dots, d_n$, such that d_i is a descendant of d_{i+1} with the label u_i . This tree is possibly the same as T if $k = n$, and otherwise is the same as T with some extra vertices with the appropriate labels added to the path. It is clear this process is also finite as there are finitely many ways to add these $n - k$ vertices.

For each such H and identified $d_1, \dots, d_n$ we construct the tree R by relabelling each $d_1, \dots, d_n$ to have labels $v_1, \dots, v_n$. In the case where a relabelling leads to a tree with a child labelled by $\perp$, we simply trim R so that this is not the case by removing excess $\perp$ labelled vertices. Each R obtained in this process forms the basis, as we add (q, R) to $\text{pb}_{\delta_n}(s)$. Note that clearly $T \leq_T H$ and $(q, R) \rightarrow (p, H)$, hence $(q, R) \in \text{Pre}(\uparrow\{s\})$.

We must also consider the case where the transition made to T' is on a path with no shared vertices to T . For this case, we add the state (q, R^*) to $\text{pb}_{\delta}(s)$, where R^* is obtained from T by simply adding an extra branch with n nodes labelled $u_1, \dots, u_n$. Similarly to before, it is clear that $(q, R^*) \in \text{Pre}(\uparrow\{s\})$.

We argue for correctness of this algorithm. Note we already proved that $\text{pb}(s) \subseteq \text{pred}(\uparrow\{s\})$, and since $\text{pred}(\uparrow\{s\})$ is upward-closed, $\uparrow\text{pb}(s) \subseteq \text{pred}(\uparrow\{s\})$. We now prove the other direction. Suppose $(q, R') \rightarrow (p, T')$ for $T \leq_T T'$. We want to show that there exists $(q, R) \in \text{pb}(s)$ such that $(q, R) \leq (q, R')$. By definition,

we can go from R' to T' by some transition $q \xrightarrow[(u_1, \dots, u_n)]{a, (v_1, \dots, v_n)} p$. Hence, there must be some set of vertices $d_1, \dots, d_n$ of T' such that each d_i is a descendant vertex of d_{i+1} , with labels u_i such that R' can be obtained from T' by relabelling the vertices with labels $v_1, \dots, v_n$, then truncating any $\perp$ branches.

Consider the path P' of T' from a leaf to a root such that it hits all vertices $d_1, \dots, d_n$. If this path does not contain any vertex of T under the mapping ϕ , then the graph R^* described above has the property that $R^* \leq_T R$, so our basis covers R . Otherwise, consider the corresponding path P in T which covers all vertices in T mapped to P' using ϕ . For each d_i not in the image of ϕ , we add to P a new vertex d'_i labelled u_i , such that the order in the tree is preserved. More concretely, this means that if $d \in P$, and d_i is a descendant of $\phi(d)$, then the new vertex corresponding to d_i is a descendant of d .

Since we considered all the ways to add new vertices to P , this new tree corresponds to some tree H generated as before. Hence, letting R be the graph obtained by relabelling $d_1, \dots, d_n$ in H with $v_1, \dots, v_n$, $(q, R) \in \text{pb}_\delta(s)$. Now, as $T \leq_T T'$ there exists a homeomorphic embedding $\phi : T \rightarrow T'$ which preserves labels. Consider the function $\phi' : R \rightarrow R'$, such that $\phi'(d_i) = d'_i$ for the new added vertices, and otherwise $\phi'(d) = \phi(d)$. This is clearly a homeomorphic embedding which preserves labels, so $R \leq_T R'$, therefore (q, R') is covered by $\text{pb}_\delta(s)$. $\square$

Corollary 3.2.3. *Emptiness of a weak f -NDCMA $\mathcal{A}$ is reducible to the covering problem for $\mathcal{S}_\mathcal{A}$, hence is decidable.*

Proof. Deciding emptiness of $\mathcal{A}$ is equivalent to asking for each accepting state $q_1, \dots, q_n \in F_G$, if there exists a path from q_0 to q_i for $1 \leq i \leq n$, and returning true if no such path exists. There exists a path from q_0 to q_i if and only if there is a transition sequence $(q_0, T_\#) \rightarrow^* (q_i, T)$ for any tree $T \in \mathcal{TL}$, where $T_\#$ is the tree with only the root, labelled $\#$. Now, $T_\# \leq_T T$ for any $T \in \mathcal{TL}$, so our problem is equivalent to deciding the coverability problem of $\mathcal{S}$ given states $s = (q_0, T_\#)$ and $t = (q_i, T_\#)$. Hence by Theorem 3.2.2, weak NDCMA emptiness is decidable. $\square$

Remark 3.2.4. The WSTS defined here is not necessarily *image-finite*. Say we have a transition with $q \xrightarrow[(u)]{a, (\perp)} p$. Then, from $s = (q, T_\#)$, where $T_\#$ is the tree

with only the root, we can go from $M_\#$ to the tree P_n , which is the path on n vertices in which the root is labelled with $\#$, leaf is labelled with u and the rest of the nodes are labelled with $\perp$. This is the case for any n , and each of these trees are distinct, hence $\mathcal{S}_A$ is not finitely branching in general. This means we cannot apply Theorem 2.2.10 for our WSTS, as this requires the WSTS to be image-finite.

3.3 Closure properties

We investigate closure properties for flexible NDCMA, proving that these automata are closed under union and intersection. This is done through well-known techniques for automata closure properties, with some minor tweaks to handle the nature of our NDCMA.

Lemma 3.3.1. *(Weak) f-NDCMA are closed under intersection.*

Proof. Let $\mathcal{A}_1 = \langle Q^{(1)}, \Sigma, \mathcal{M}^{(1)}, \delta^{(1)}, q_0^{(1)}, \mathcal{M}_L^{(1)}, F_G^{(1)} \rangle$, $\mathcal{A}_2 = \langle Q^{(2)}, \Sigma, \mathcal{M}^{(2)}, \delta^{(2)}, q_0^{(2)}, \mathcal{M}_L^{(2)}, F_G^{(2)} \rangle$ be two f-NDCMA. Without loss of generality, we can assume that the state and label sets of the two automata are disjoint. We also let $\delta^{(1)} = \bigcup_{i=0}^{k_1} \delta_i^{(1)}$ and $\delta^{(2)} = \bigcup_{i=0}^{k_2} \delta_i^{(2)}$.

We will construct an automaton $\mathcal{B}$ which will simulate $\mathcal{A}_1$ and $\mathcal{A}_2$ in tandem using the cross product construction. In particular, $\mathcal{B} = \langle Q^{(1)} \times Q^{(2)}, \Sigma, (\mathcal{M}^{(1)} \cup \{\perp_1\}) \times (\mathcal{M}^{(2)} \cup \{\perp_2\}), \Delta, (q_0^{(1)}, q_0^{(2)}), (\mathcal{M}_L^{(1)} \cup \{\perp_1\}) \times (\mathcal{M}_L^{(2)} \cup \{\perp_2\}), F_G^{(1)} \times F_G^{(2)} \rangle$. Now, for each pair of transitions

$$p \xrightarrow[(u_1, \dots, u_n)]{a, (v_1, \dots, v_n)} q \quad p' \xrightarrow[(u'_1, \dots, u'_m)]{a, (v'_1, \dots, v'_m)} q'$$

in $\mathcal{A}_1, \mathcal{A}_2$ respectively, we will add multiple transitions to Δ . We note that when transitioning in tandem, the input data value d needs to have the label u_1, v_1 respectively, while the ancestors of d chosen for each transition could be different. We need to consider all the different ways to interleave the labels, and hence we consider strings $s = ((a_1, a'_1), (a_2, a'_2), \dots, (a_l, a'_l))$ for which:

- $1 = i_1 < i_2 < \dots < i_n \leq l$, and $a_{i_t} = v_t$ for $1 \leq t \leq n$, $a_t = *$ otherwise;
- $1 = j_1 < j_2 < \dots < j_m \leq l$, and $a'_{j_t} = v'_t$ for $1 \leq t \leq m$, $a'_t = *$ otherwise;

- $l = \max(i_n, j_m)$.

This essentially gives us all the possible ways we could interleave the labels for both data values, with $*$ acting as a marker for any possible value allowed. We further consider strings s' where each v_t and v'_t is replaced with u_t and u'_t . Then to Δ we add the transition $(p, p') \xrightarrow[s']{a, s} (q, q')$.

Note that the ' $*$'s in the top stand for arbitrary values, but the ' $*$'s at the bottom of the transition represent keeping the values unchanged. However, there is one exception when this can't happen, namely when $*$ is used to represent $\perp$. This is because definitionally, we have to give a data value a label once we see it for the first time. To handle this, we have special values $\perp_1$ and $\perp_2$ which we will use to replace $\perp$. We further modify the transitions such that any possible transitions with $\perp$ can be made with the respective $\perp_i$ value, and add these values to the accepting local labels to preserve acceptance.

Our automaton $\mathcal{B}$ simulates the two automata in tandem, where the labels of $\mathcal{A}_1$ is stored in the left hand of the cross product, and those of $\mathcal{A}_2$ are stored in the right hand of the cross product. This way, for all the data values to be locally accepting for both automata, these need to be contained in $(\mathcal{M}_L^{(1)} \cup \{\perp_1\}) \times (\mathcal{M}_L^{(2)} \cup \{\perp_2\})$ as desired. Hence, whenever $w \in \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$, $w \in \mathcal{L}(\mathcal{B})$. We note that this construction preserves weakness as if $\mathcal{M}_L^{(i)} = \mathcal{M}^{(i)}$ then $(\mathcal{M}_L^{(1)} \cup \{\perp_1\}) \times (\mathcal{M}_L^{(2)} \cup \{\perp_2\}) = (\mathcal{M}^{(1)} \cup \{\perp_1\}) \times (\mathcal{M}^{(2)} \cup \{\perp_2\})$. $\square$

We also present the simpler result that flexible NDCMA are closed under union. For this result we use the typical construction of adding a new start state which acts non-deterministically like the start state of the two automata, capturing the behaviour of both automata. Note that we could have also shown that ϵ -transitions do not add expressivity to the automata model, as we do later on for a different NDCMA formulation.

Lemma 3.3.2. *(Weak) f-NDCMA are closed under union.*

Proof. Let $\mathcal{A}_1 = \langle Q^{(1)}, \Sigma, \mathcal{M}^{(1)}, \delta^{(1)}, q_0^{(1)}, \mathcal{M}_L^{(1)}, F_G^{(1)} \rangle$, $\mathcal{A}_2 = \langle Q^{(2)}, \Sigma, \mathcal{M}^{(2)}, \delta^{(2)}, q_0^{(2)}, \mathcal{M}_L^{(2)}, F_G^{(2)} \rangle$ be two flexible NDCMA. Without loss of generality, we can assume that the state and label sets of the two automata are disjoint.

We construct $\mathcal{B} = \langle Q^{(1)} \cup Q^{(2)} \cup \{q_0\}, \Sigma, \mathcal{M}^{(1)} \cup \mathcal{M}^{(2)}, \Delta, q_0, \mathcal{M}_L^{(1)} \cup \mathcal{M}_L^{(2)}, F_G \rangle$, letting $F_G = F_G^{(1)} \cup F_G^{(2)} \cup \{q_0\}$ if $q_0^{(i)} \in F_G^{(i)}$ for some $i \in \{1, 2\}$ and $F_G = F_G^{(1)} \cup F_G^{(2)}$ otherwise. Essentially our new start state q_0 is accepting if and only if one of the automata's start states were accepting.

Then, we let $\Delta = \delta^{(1)} \cup \delta^{(2)} \cup \delta^{\text{init}}$, where δ^{init} consists of all transitions

$$q_0 \xrightarrow[(u_1, \dots, u_n)]{a, (v_1, \dots, v_n)} p \quad \text{when} \quad q_0^{(i)} \xrightarrow[(u_1, \dots, u_n)]{a, (v_1, \dots, v_n)} p \text{ for any } i.$$

This means that q_0 will act like both $q_0^{(1)}$ and $q_0^{(2)}$. The first transition will establish which automaton we will be in as the state spaces for the two are disjoint, and there are no transitions back to q_0 so this construction does not interfere with the execution of the automaton. Hence, $w \in \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$ if and only if $w \in \mathcal{L}(\mathcal{B})$. $\square$

Unfortunately, flexible NDCMA are not closed under complement. This can be seen immediately from the fact that they coincide with CMA when letting $\delta = \delta_1$, and CMA are not closed under complement [3].

Lemma 3.3.3. *(Weak) f-NDCMA are not closed under complement.*

Remark 3.3.4. In [14], to obtain closure under complement for CMA the authors consider deterministic weak class memory automata. For this model, every word w corresponds to exactly one run, so flipping the accepting and non-accepting states, we can obtain the complement of the accepted language.

However, the same trick cannot be done for our flexible NDCMA, as the model itself is inherently non-deterministic. Suppose for example, we have the transitions $p \xrightarrow[v, u_1]{a, (v, u_1)} q_{\text{accept}}$ and $p \xrightarrow[v, u_2]{a, (v, u_2)} q_{\text{reject}}$, and a data value d at level 2, where $\text{pred}(d) = d_1$ and $\text{pred}^2(d) = d_2$. Let $f(d) = v$, $f(d_1) = u_1$ and $f(d_2) = u_2$. It is clear that given input (a, d) , the automaton can choose either one of these transitions. However, this automaton can still be ‘deterministic’ with respect to the definition as this only enforces that given the same *labels* and state, the automaton transitions the same, and our transitions have different labels.

We would like some stronger notion of determinism, which implies that each word w corresponds to only one run of the automaton, however our framework limits us here. Moreover, it is easy to see that using f-NDCMA is difficult when

we want to be more deliberate about our transitions. For example, if we want new data values to have their parents initialised, there is no way of enforcing this for our flexible NDCMA. Referring back to our initial motivation of representing programs with loops, this is even more troublesome as we generally want to have available to us the names within scope, not all possible names. For this reason, in the next chapter we will explore an alternate model of NDCMA which allows us more control.

Chapter 4

Rigid NDCMA

In this chapter we introduce *rigid NDCMA*, which will allow us more control over the view of a node. Flexible NDCMA, introduced in Chapter 3 match sequences of labels to a sequence of labels for a selection of not necessarily consecutive ancestors. In contrast, in rigid NDCMA these ancestors will be specified using a special ‘memory’, which will be maintained for each node. This memory is initialised for a new node based on the state and its parent’s memory, and remains fixed throughout the execution. This is not to be confused with the class memory function, which remembers labels from a set $\mathcal{M}$ to data values. To avoid conflicts of terminology, we refer to the node’s ancestor memory as *memory*, and the values of the class memory function as *labels*. We follow a similar process as in Chapter 3, proving decidability of emptiness using WSTS theory and exploring closure properties. Additionally, we include a section on Petri nets, these will be useful to prove some undecidability results.

4.1 Introducing r-NDCMA

Definition 4.1.1. Let k be a fixed natural number. A k -*rigid Nested Data Class Memory Automaton* (*r-NDCMA*) over the alphabet $\Sigma \times \mathcal{D}$, where $\mathcal{D}$ is a nested data set is a tuple $\mathcal{A} = \langle Q, \Sigma, \mathcal{M}, \delta, \sigma, q_0, \mathcal{M}_L, F_G \rangle$ where

- Q is a finite set of states;
- Σ is a finite alphabet of tags;

- $\mathcal{M}$ is a finite set of labels;
- q_0 is the initial state;
- $F_G \subseteq Q$ is the set of globally accepting states;
- $\mathcal{M}_L \subseteq \mathcal{M}$ is the set of locally accepting labels;
- δ, σ are transition functions for initialised and uninitialised data respectively, where $\delta = \bigcup_{i=0}^k \delta_i$, $\sigma = \bigcup_{i=-1}^k \sigma_i$ and

$$\delta_i : Q \times \Sigma \times \mathcal{M}^{i+1} \rightarrow \mathcal{P}(Q \times \mathcal{M}^{i+1})$$

$$\sigma_i : Q \times \Sigma \times \mathcal{M}^{i+1} \rightarrow \mathcal{P}(Q \times \mathcal{M}^{i+2} \times \mathbb{N})$$

such that for any $(p, \vec{v}, j)$ in the range of σ_i , $j \leq \min(i+1, k)$.

If for every (q, t, n) in the image of σ_i , $n = i+1$, then we say $\mathcal{A}$ is a *bounded NDCMA*, and in fact these NDCMA coincide with those defined in [14]. This is because for these automata, each node has access to all its ancestors — which corresponds to always having a maximal memory size. Moreover, borrowing notation from [14], we say that $\mathcal{A}$ is *deterministic* if each set in the image of δ, σ is a singleton, and $\mathcal{A}$ is *weak* if all labels are locally accepting, that is $\mathcal{M}_L = \mathcal{M}$. We sometimes omit $\mathcal{M}_L$ in the automaton description when $\mathcal{A}$ is weak. Note that if $\mathcal{A}$ is deterministic, we can represent its transition functions as functions $\delta_i : Q \times \Sigma \times \mathcal{M}^{i+1} \rightarrow Q \times \mathcal{M}^{i+1}$, $\sigma_i : Q \times \Sigma \times \mathcal{M}^{i+1} \rightarrow Q \times \mathcal{M}^{i+2} \times \mathbb{N}$.

We use $q \xrightarrow[\substack{v'_1, \dots, v'_{i+1}}]{a, (v_1, \dots, v_{i+1})} q'$ to denote $(q', (v'_1, \dots, v'_{i+1})) \in \delta_i(q, a, (v_1, \dots, v_{i+1}))$, and $q \xrightarrow[\substack{(v', v'_1, \dots, v'_{i+1}), m=j}]{a, (\perp, v_1, \dots, v_{i+1})} q'$ to denote $(q', (v', v'_1, \dots, v'_{i+1}), j) \in \sigma_i(q, a, (v_1, \dots, v_{i+1}))$.

Definition 4.1.2. A *configuration* for a k -rigid NDCMA $\mathcal{A} = \langle Q, \Sigma, \mathcal{M}, \delta, \sigma, q_0, \mathcal{M}_L, F_G \rangle$ over $\Sigma \times \mathcal{D}$ is a pair (q, f, m) , where $q \in Q$, $f : D \rightarrow \mathcal{M}_\perp$ and $m : D \rightarrow D^{\leq k} = \bigcup_{i=0}^k D^i$ such that f is a *class memory function*, that is $f(d) \neq \perp$ for only finitely many d . The *initial configuration* of the automaton is (q_0, f_0, m_0) , where $f_0(d) = \perp$ and $m_0(d) = \epsilon$ for all $d \in D$. A configuration (q, f, m) is *accepting* if $q \in F_G$ and $f(D) \subseteq \mathcal{M}_L \cup \{\perp\}$.

The automaton can transition from configuration (q, f, m) to (q', f', m') on reading input (a, d) where $a \in \Sigma$, $d \in D$ if

1. If $f(d) \neq \perp$, then the following conditions hold

- (a) $m(d) = (d_1, d_2, \dots, d_i)$,
- (b) $q \xrightarrow[(v_0, v_1, \dots, v_i)]{a, (f(d), f(d_1), \dots, f(d_i))} q'$, that is, $(q', (v_0, v_1, \dots, v_i)) \in \delta_i(q, a, (f(d), f(d_1), \dots, f(d_i)))$,
- (c) $m = m'$ and $f' = f[d \mapsto v_0, d_1 \mapsto v_1, \dots, d_i \mapsto v_i]$.

2. If $f(d) = \perp$, then the following conditions hold

- (a) $d_0 = f(\text{pred}(d)) \neq \perp$ and $m(d_0) = (d_1, d_2, \dots, d_i)$, or $\text{pred}(d)$ is undefined and $i = -1$ (to deal with the case when d is a root).
- (b) $q \xrightarrow[(v, v_0, v_1, \dots, v_i), m=j]{a, (\perp, f(d_0), f(d_1), \dots, f(d_i))} q'$, that is, $(q', (v, v_0, v_1, \dots, v_i), j) \in \sigma_i(q, a, (f(d_0), f(d_1), \dots, f(d_i)))$,
- (c) $m = m'[d \mapsto (d_{i-j+1}, \dots, d_i)]$ and $f' = f[d \mapsto v, d_0 \mapsto v_0, \dots, d_i \mapsto v_i]$.

In words, to transition on an initialised value d , we match the labels from $\mathcal{M}$ of d , as well as the labels of the data values from its memory $m(d)$, updating the labels of d and its memory according to the transition. For an uninitialised value d , we match with the labels of its parent $\text{pred}(d)$ and its parents' memory $m(\text{pred}(d))$ and update these labels accordingly. The memory of d is now initialised, and depends on the j specified in the transition. Its memory becomes the length j suffix of the list $\text{pred}(d), m(\text{pred}(d))$.

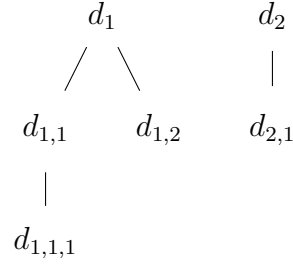
Note that a data value's memory never exceeds length k as we enforce this in the definition of σ . The memory of a data value increases from its parents whenever it includes the parent in the memory ($j = i + 1$), and decreases when it forgets some prefix of the parent's memory ($j < i$).

A *run* $(q_0, f_0, m_0), (q_1, f_1, m_1), \dots, (q_n, f_n, m_n)$ is a sequence of configurations such that (q_0, f_0, m_0) is the initial configuration and $(q_{i+1}, f_{i+1}, m_{i+1})$ is valid next configuration from (q_i, f_i, m_i) . Moreover, this run is *accepting* if and only if (q_n, f_n, m_n) is an accepting configuration. For $w \in (\Sigma \times \mathcal{D})^*$, we say that $\mathcal{A}$ accepts w if and only if there exists an accepting run in which the input at each step i is $w[i]$. We let the language of $\mathcal{A}$, $\mathcal{L}(\mathcal{A}) \subseteq (\Sigma \times \mathcal{D})^*$ be the set of words accepted by $\mathcal{A}$.

Example 4.1.3. We give an example of a weak 1-rigid NDCMA, with only one state, $\mathcal{FA} = \langle \{q_0\}, \Sigma, \mathcal{M}, \delta, \sigma, q_0, \mathcal{M}_L, F_G \rangle$, where $\Sigma = \{\text{spawn}, \text{activate}, \text{deactivate}\}$, $\mathcal{M} = \mathcal{M}_L = \{\text{active}, \text{inactive}, 0, 1\}$, and δ is made up of transitions

$$\begin{array}{lcl}
(t_{\text{spawnR}}) & \left| \begin{array}{c} q_0 \xrightarrow[\substack{(0), m=0}]{\text{spawn}, (\perp)} \\ q_0 \xrightarrow[\substack{(\text{inactive}, 0), m=1}]{\text{spawn}, (\perp, 0)} \\ q_0 \xrightarrow[\substack{(\text{inactive}, \text{inactive}, 0), m=1}]{\text{spawn}, (\perp, \text{inactive}, 0)} \\ q_0 \xrightarrow[\substack{(\text{active}, 1)}]{\text{activate}, (\text{inactive}, 0)} \\ q_0 \xrightarrow[\substack{(\text{inactive}, 1)}]{\text{deactivate}, (\text{active}, 1)} \end{array} & q_0, \\
(t_{\text{spawnA}}) & & q_0, \\
(t_{\text{spawnB}}) & & q_0, \\
(t_{\text{act}}) & & q_0, \\
(t_{\text{deact}}) & & q_0.
\end{array}$$

Consider the data values $d_1, d_2, d_{1,1}, d_{1,1,1}, d_{1,2}, d_{2,1} \in D$ with predecessor relationships as specified in the diagram below. We will consider some simple runs



using these data values. We start with $f(d_i) = \perp$, for each i . Suppose we see the input (spawn, d_1) . Since d_1 is a level 0 value, we transition with t_{spawnR} , setting $f(d_1) = 0$, and $m(d_1) = \epsilon$. Next, we see as input $(\text{spawn}, d_{1,1})$. Since $d_{1,1}$ is not yet initialised, we look at its parent, d_1 , and its parent's memory, which in this case is the empty tuple. Since $f(d_1) = 0$, we can transition with t_{spawnA} , giving $d_{1,1}$ the value *inactive*. Moreover, $m = 1$, so $|m(d_{1,1})| = 1$. We take the length 1 suffix of the tuple $d_1, m(d_1)$, which in this case is just (d_1) , hence $m(d_{1,1}) = (d_1)$

Now, if we see input $(\text{spawn}, d_{1,1,1})$, this value is again new as $f(d_{1,1,1}) = \perp$, so we look to its parent, $d_{1,1}$. As $m(d_{1,1}) = d_1$, we need to match with the values of

$$(f(d_{1,1,1}), f(d_{1,1}), f(d_1)) = (\perp, \text{inactive}, 0),$$

which are in fact the values specified in t_{spawnB} , so we can transition using this. This sets $f(d_{1,1,1}) = \textit{inactive}$, and keeps the other labels the same. Moreover, we have $m = 1$, meaning $|m(d_{1,1,1})| = 1$. We take the length 1 suffix of $d_{1,1}$, $m(d_{1,1}) = (d_{1,1}, d_1)$ and hence we forget the parent value $d_{1,1}$, and simply set $m(d_{1,1,1}) = (d_1)$.

Suppose that next we see the value $(\textit{activate}, d_{1,1,1})$. We match with the label of $d_{1,1,1}$ as well as the labels of the memory of $d_{1,1,1}$, which in this case is just (d_1) , hence we must match with $(f(d_{1,1,1}), f(d_1)) = (\textit{inactive}, 0)$. This does in fact match the labels in t_{act} , so we can transition, updating $f(d_1) = 1$, and $f(d_{1,1,1}) = \textit{active}$.

Now, it should be clear that we cannot transition with input $(\textit{activate}, d_{1,1})$ or even input $(\textit{spawn}, d_{1,2})$ as $f(d_1) = 1$, hence this would not match any of our patterns. We are however allowed to transition with $(\textit{spawn}, d_2)$, as this node does not have reference to d_1 . We can also transition given input $(\textit{deactivate}, d_{1,1,1})$, and this sets $f(d_{1,1,1}) = \textit{inactive}$ and $f(d_1) = 0$. From this point we can now transition with $(\textit{spawn}, d_{1,2})$.

Given this, we can see that the data word $(\textit{spawn}, d_1), (\textit{spawn}, d_{1,1}), (\textit{spawn}, d_{1,1,1}), (\textit{activate}, d_{1,1,1}), (\textit{spawn}, d_2), (\textit{deactivate}, d_{1,1,1}), (\textit{spawn}, d_{1,2})$ is a valid word in $\mathcal{L}(\mathcal{A})$. Because of the way the automaton is defined, each new data value will keep track of the level 0 value, corresponding to the root. If a data value is activated, this means that no more new data values can be spawned with the same root until it is deactivated, but we can still spawn new roots and data values with different roots to the activated value.

It is clear that these automata can capture unbounded behaviour, as the above automaton would accept the word $(\textit{spawn}, d_1), \dots, (\textit{spawn}, d_n)$, where $\text{pred}(d_{i+1}) = d_i$. Moreover, we have more control using these automata than the flexible-NDCMA, as we are specifying in particular the useful ancestors for a data value. Here we have modelled a scenario similar to that of avoiding a deadlock, where only one data value from a given tree can be active at a time. Moreover, we will apply these versatile models further in Chapter 5 to represent traces of a programming language.

4.2 WSTS for r-NDCMA

As was done for bounded NDCMA and f-NDCMA, we will define a well-structured transition system to represent the states and transitions of an r-NDCMA. To do this, we first define some machinery which will allow us to represent the memory of an r-NDCMA using a bounded graph.

From a function $m : D \rightarrow (D)^{\leq k}$ on some finite set D , we construct a graph, which we will call a memory graph. When m is the corresponding function in a r-NDCMA configuration, this graph will be a tree corresponding to the underlying memory structure.

Definition 4.2.1. Given a finite set D and $m : D \rightarrow (D)^{\leq k}$, we construct the *memory graph* $(D \cup \{r\}, \text{pred}')$ inductively as follows:

1. If $m(d) = \epsilon$, $\text{pred}'(d) = r$, where r is a distinguished symbol for the root
2. If $m(d) = (d_1, \dots, d_n)$, $\text{pred}'(d) = d_1$.

Given a class memory function $f : D \rightarrow \mathcal{M}_\perp$ over a nested data set $\mathcal{D}$, we say the memory graph of $\mathcal{D}$, denoted $m(\mathcal{D})$ is the memory graph over the finite subset $F \subseteq D$ such that $F = \{d \in D : f(d) \neq \perp\}$. Moreover, this graph can be seen as a labelled graph, with labels from $\mathcal{M}$ given by f , and a distinguished label for the root r .

While this definition does not enforce that a memory graph is indeed a tree, this will be the case when the function $m : D \rightarrow (D)^{\leq k}$ is the memory function of an r-NDCMA. In this case, we will sometimes refer to $(D \cup \{r\}, \text{pred}')$ as a *memory tree*. We define the set of trees $m_k(\mathcal{T})$ such that each tree corresponds to some memory graph resulting from a valid configuration of the automaton.

Definition 4.2.2. Given a nested data set $\mathcal{D} = (D, \text{pred})$, we define the set of all memory trees of size k , $m_k(\mathcal{T})$ to be the set of trees on a finite vertex set $F \subseteq D$ along with a distinguished root vertex r such that

- $d \in F$ implies $\text{pred}(d) \in F$;
- $(\text{pred}')^i(d) = r$ for some $i \leq k + 1$;

- $\text{pred}'(d) = r$ or $\text{pred}'(d) = \text{pred}^t(d)$ for some $t \in \mathbb{N}$.

Given some finite set S , we further define the labelled analogue of these trees $m_{k,S}(\mathcal{T})$ such that r is labelled by a special symbol $\# \notin S$, and the rest of the vertices are given a label from S .

Moreover, given a tree $T \in m_{k,\mathcal{M}}(\mathcal{T})$ with domain F , we define the functions $f_T : D \rightarrow \mathcal{M}_\perp$ and $m_T : D \rightarrow D^k$ as

- $f_T(d) = l$ if $d \in F$ and d has label l in T , and $f_T(d) = \perp$ otherwise;
- $m_T(d) = (\text{pred}'(d), \dots, (\text{pred}')^i(d))$ if $d \in F$ and $(\text{pred}')^{i+1}(d) = r$, and $m_T(d) = \epsilon$ otherwise.

Clearly $m_{k,\mathcal{M}}(\mathcal{T})$ consists of bounded trees of depth $k + 2$. There is a one to one correspondence between a memory graph in $m_{k,\mathcal{M}}(\mathcal{T})$ on a nested data set $\mathcal{D}$ and the functions f, m in a k -rigid NDCMA. We see this through the following lemma.

Lemma 4.2.3. *For any valid configuration (q, f, m) of a k -rigid NDCMA on data set $\mathcal{D}$ with labels from $\mathcal{M}$, the memory graph $m(\mathcal{D})$ labelled by the function f is in $m_{k,\mathcal{M}}(\mathcal{T})$.*

Moreover, for each $d \in \mathcal{D}$, $|m(d)| = i$ implies $m(d) = (\text{pred}'(d), \dots, (\text{pred}')^i(d))$, and $(\text{pred}')^{i+1}(d) = r$. Therefore $f = f_{m(\mathcal{D})}$ and $m = m_{m(\mathcal{D})}$.

Proof. First note that by definition, for populated d , $m(d) = (d_1, \dots, d_k, \dots, d_n)$ implies $m(d_k) = (d_{k+1}, \dots, d_n)$. This follows from the fact that we always take the suffix of the parent and its memory. This implies first that for each d with $|m(d)| = i > 0$, $|m(\text{pred}'(d))| = i - 1$. Clearly then, our graph does not have cycles. Moreover, if $|m(d)| = 0$ we have by definition that $\text{pred}'(d) = r$, so our graph is a rooted tree. From this it also immediately follows that $|m(d)| = i$ implies $m(d) = (\text{pred}'(d), \dots, (\text{pred}')^i(d))$, and $(\text{pred}')^{i+1}(d) = r$. The last part of the lemma follows from the definition of f_T, m_T and the fact that m_0 assigns every value d to the empty tuple ϵ . $\square$

Lemma 4.2.4. *Fix a nested data set $\mathcal{D}$, and let $\mathcal{A}$ be an r -NDCMA. $\mathcal{S}_{\mathcal{A}} = \langle S, \rightarrow, \leq \rangle$ as defined below is a WSTS.*

- $S = Q \times m_{k,\mathcal{M}}(\mathcal{T})$,
- $(q, T) \rightarrow (q', T')$ if and only if $(q', f_{T'}, m_{T'})$ is a valid successor configuration from (q, f_T, m_T) ,
- $(q, T) \leq (q', T')$ if and only if $q = q'$ and $T \leq_C T'$ as in Definition 2.1.12, where $(A, \leq) = (\mathcal{M} \cup \{\#\}, =)$.

Proof. First note it is clear that $\leq$ is a wqo on S , using the fact that $\leq_C$ is a wqo on trees, $=$ is a wqo on finite sets and the product ordering of wqos is a wqo.

Now, we need to prove the upward compatibility condition, which states that $(q, T) \rightarrow (q', T')$ and $(q, T) \leq (p, R)$ implies that there exists some $(p', R') \geq (q', T')$ such that $(p, R) \rightarrow (p', R')$. Now, suppose $(q', f_{T'}, m_{T'})$ is a valid successor configuration from (q, f_T, m_T) using the transition

$$q \xrightarrow[a, (u_0, u_1, \dots, u_i)]{(v_0, v_1, \dots, v_i)} q'.$$

This means there exists some initialised data value d with $m_T(d) = (d_1, d_2, \dots, d_i)$ such that $(f(d), f(d_1), \dots, f(d_i)) = (v_0, v_1, \dots, v_i)$ and $f_{T'}$ is the same as f_T with the vertices $d, d_1, \dots, d_i$ relabelled $u_0, u_1, \dots, u_i$.

Since $T \leq_C R$, there exists an injective tree homomorphism $\phi : T \rightarrow R$. This implies that exists an initialised data value $d' = \phi(d)$ in R with $m_R(d') = (d'_1, d'_2, \dots, d'_i)$ such that $d', d'_1, \dots, d'_i$ are labelled $v_0, v_1, \dots, v_i$. Note we have that $|m_R(d')| = i$ since it is implicit that $\text{pred}'(d_i) = r$ and the root is always mapped to the root. Furthermore $p = q$, so we can transition from (p, m_R, f_R) in the same way as we transitioned from (q, m_T, f_T) to some new configuration $(q', m_{R'}, f_{R'})$. Now, R' has the same vertices as R , except $d', d'_1, \dots, d'_i$ are labelled $u_0, u_1, \dots, u_i$. From this it is clear that $T' \leq_C R'$ using the same embedding ϕ as for T .

Now if instead we transition given an uninitialised value d , first we note that $d_1 = \text{pred}(d)$ must be in T as we can only initialise data which has an initialised parent. Then, our transition is determined by the labels of d_1 and $m_T(d_1) = (d_2, \dots, d_i)$. Suppose we transition with

$$q \xrightarrow[(u_0, u_1, \dots, u_i), m=j]{a, (\perp, v_1, \dots, v_i)} q'$$

where $(f(d_1), \dots, f(d_i)) = (v_1, \dots, v_i)$. T' is such that $m_{T'}(d) = (d_{i-j+1}, \dots, d_i)$ and $(d, d_1, \dots, d_i) = (u_0, u_1, \dots, u_i)$.

As before, we have that $\phi(d_1) = d'_1$ where d'_1 has label v_1 in R , and hence $m_R(d'_1) = (d'_2, \dots, d'_i)$ with labels $v_2, \dots, v_n$. This means that we can transition the same way as from T to $(q', m_{R'}, f_{R'})$, adding a new node $d' \in D$ such that $\text{pred}(d') = d_0$ at level $i - j + 1$ and labelling the vertices with $u_0, u_1, \dots, u_i$. Now ϕ mapping T to R can be adapted to be a tree homomorphism from T' to R' where we map the new data value d to d' . $\square$

Theorem 4.2.5. *Let $\mathcal{A}$ be an r -NDCMA. The covering problem for $\mathcal{S}_{\mathcal{A}}$ is decidable.*

Proof. By Theorem 2.2.11, we are left to prove that $\mathcal{S}_{\mathcal{A}}$ has an effective pred-basis and decidable $\leq$. It is clear that $\leq$ is decidable as $T_1 \leq_C T_2$ can be decided by iterating through all possible injections from T_1 to T_2 and checking if at least one of them is a tree homomorphism. The fact that $\mathcal{S}_{\mathcal{A}}$ has an effective pred-basis follows very similarly to [12] and Theorem 3.2.2. The underlying idea is to identify parts of the memory graphs that could be the result of applying a transition, and reversing this process to obtain what the graph would have looked like before the transition.

Let $s = (p, T)$ be fixed. We establish an algorithm for calculating $\text{pb}(s)$ such that $\uparrow \text{pb}(s) = \uparrow \text{Pre}(\uparrow \{s\}) = \text{Pre}(\uparrow \{s\})$. We do this by finding a basis $\text{pb}_{\delta_i}(s)$ and $\text{pb}_{\sigma_i}(s)$ for each i and letting $\text{pb}(s) = \bigcup_{i=0}^k \text{pb}_{\delta_i}(s) \cup \bigcup_{i=-1}^k \text{pb}_{\sigma_i}(s)$. First note in any case if p is not in the range of the transition function, we set the basis to be empty, as each $s' = (p', T')$ such that $s \leq s'$ has $p' = p$ hence is unreachable using the transition function.

Let $\delta_i : Q \times \Sigma \times (\mathcal{M})^{i+1} \rightarrow \mathcal{P}(Q \times (\mathcal{M})^{i+1})$ and let

$$q \xrightarrow[(u_0, u_1, \dots, u_i)]{a, (v_0, v_1, \dots, v_i)} p$$

be a particular transition in δ_i . For $0 \leq j \leq i+1$, let P_j be the set of all tuples size $j+1$ of vertices $(d_j, \dots, d_i, r)$ such that $\text{pred}'(d_k) = \text{pred}'(d_{k+1})$ for all $0 \leq k \leq j$, and d_k has label u_k . Note if $j = i+1$, we are considering the tuple with no data values, (r) , and any references to d_j will refer to r . We iterate through all such possible tuples, noting that the set P_j is finite since the graph itself is.

Consider the graph $H \in m_{k,\mathcal{M}}(\mathcal{T})$ obtained from T by appending fixed arbitrary data values $(c_0, \dots, c_{j-1})$, each c_k labelled by u_k , not in the domain of T , $\text{pred}'(c_k) = c_{k+1}$ for $0 \leq k \leq i-j-1$ and $\text{pred}'(c_{i-j}) = d_j$. Note if $j = 0$, we add no elements and skip this step, letting $H = T$. For H to actually be in $m_{k,\mathcal{M}}(\mathcal{T})$, these c_k s must be descendants of each other in terms of (D, pred) and their parent must be initialised, but we can easily ensure this is true by letting c_{i-j} be an uninitialised child of d_j and c_k a child of c_{k+1} . Note that the choice here is arbitrary, but a choice does need to be made as otherwise we would be adding infinite elements to the pred-basis. Now, let $R \in m_{k,\mathcal{M}}(\mathcal{T})$ be the graph obtained by relabelling the vertices $c_0, c_1, \dots, c_{i-j}, d_j, \dots, d_i$ with labels $v_0, \dots, v_i$, and add (q, R) to $\text{pb}_{\delta_i}(s)$. It is immediate that $(q, R) \in \text{Pre}(\uparrow\{s\})$ as $(q, R) \rightarrow (p, H)$, and clearly $T \leq_C H$.

Now if $s \leq s'$, that is $s' = (p, H)$ with $T \leq_C H$, then there is some injective tree homomorphism $\phi : T \rightarrow H$. Suppose $(q, R') \rightarrow (p, H)$ using the above transition on data values $(c_0, \dots, c_i)$. Letting j be the number of vertices from $(c_0, \dots, c_i)$ which are not in the range of ϕ , it is clear that $(d_{j+1}, \dots, d_i, r) \in P_j$ as defined above. Then, we have that $(q, R) \in \text{pb}_{\delta_i}(s)$, where R is the graph obtained by adding new vertices $c_0, \dots, c_j$ with the labels $v_0, \dots, v_j$. Adapting the mapping ϕ , mapping the new vertices appropriately, it is clear that $R \leq_C R'$ as required.

Let $\sigma_i : Q \times \Sigma \times (\mathcal{M})^{i+1} \rightarrow \mathcal{P}(Q \times (\mathcal{M})^{i+2} \times \mathbb{N})$ and consider the particular transition

$$p \xrightarrow[(u_0, u_1, \dots, u_{i+1}), m=n]{a, (v_1, \dots, v_{i+1})} q.$$

We follow the exact same process as we had for δ , except now we have to take into account the new initialised vertex. Consider again the set P_j of vertices $d_j, \dots, d_{i+1}$ with labels $u_j, \dots, u_{i+1}$ defined as before. When the size of the tuple is at least n , that is $(i+1) - j + 1 \geq n$, we additionally consider for each tuple all the possible extra vertices d with label u_0 such that $\text{pred}(d) = d_{i+n+2}$, meaning all the possible vertices with memory size n .

As before, we pick $c_1, \dots, c_{j-1}$ with labels $u_0, \dots, u_{j-1}$ to complete the path. Moreover, we must also pick an extra vertex c_0 with memory size n if we haven't already picked such a vertex. We call this resulting graph H . Again, we ensure $H \in m_{k,\mathcal{M}}(\mathcal{T})$ by picking appropriate data values, and we ensure these data values

are disjoint from those in the range of T . Then, we obtain $R \in m_{k,\mathcal{M}}(\mathcal{T})$ by relabelling $c_0, \dots, c_{j-1}, d_j, \dots, d_{i+1}$ with labels $v_0, \dots, v_i$ and removing the vertex at level n . We put (q, R) in $\text{pb}_{\sigma_i}(s)$.

It is clear that $(q, R) \rightarrow (p, H)$ and $T \leq_C H$ for any H generated above. We also see that for any $T \leq_C H'$, if $(q, R') \rightarrow (p, H')$ using σ_i , we can obtain $R \leq_C R'$ such that $(q, R') \in \text{pb}_{\sigma_i}(s)$ by identifying the common vertices of T and H' , adding appropriate $c_1, \dots, c_j$, relabelling these and removing the extra node at level n if necessary.

□

Corollary 4.2.6. *Emptiness of a weak r -NDCMA $\mathcal{A}$ is reducible to the covering problem for $\mathcal{S}_{\mathcal{A}}$, hence is decidable.*

Proof. Deciding emptiness of an r -NDCMA $\mathcal{A}$ is equivalent to asking for each accepting state $q_1, \dots, q_n \in F_G$, if there exists a path from q_0 to q_i for $1 \leq i \leq n$, and returning true if no such path exists. There exists a path from q_0 to q_i if and only if there is a transition sequence $(q_0, T_{\#}) \rightarrow^* (q_i, T)$ for any tree $T \in m_{k,Q}(\mathcal{T})$, where $T_{\#}$ is just the memory tree with only the root, labelled $\#$. Now, $T_{\#} \leq_T T$ for any $T \in m_{k,Q}(\mathcal{T})$, so our problem is equivalent to deciding the coverability problem of $\mathcal{S}$ given states $s = (q_0, T_{\#})$ and $t = (q_i, T_{\#})$. Hence by Theorem 4.2.5, weak r -NDCMA emptiness is decidable.

□

While less applicable to automata theory, we also have that the WSTS described above can be transformed into one which is image-finite and post-effective, and hence we can decide termination as well.

Theorem 4.2.7. *Let $\mathcal{A}$ be a n r -NDCMA. The termination problem for $\mathcal{S}_{\mathcal{A}}$ is decidable.*

Proof. By Theorem 2.2.10, we have to prove that $\mathcal{S}_{\mathcal{A}}$ is image-finite and post-effective. Unfortunately this is not immediately the case as our state space is $Q \times m_{k,\mathcal{M}}(\mathcal{T})$, where our trees are over vertices of D , hence for each possible configuration of the memory graph there are infinitely many trees in $m_{k,\mathcal{M}}(\mathcal{T})$.

Let $\mathcal{TL}_{\mathcal{M},k}$ be the set of unordered trees of depth k with root labelled $\#$, and other nodes labelled with $\mathcal{M}$, and for $T \in m_{k,\mathcal{M}}(\mathcal{T})$ let $\text{unord}(T)$ be tree obtained from T , ignoring order and vertices. Consider the alternate definition $\mathcal{S}'_{\mathcal{A}}$

- $S = Q \times \mathcal{TL}_{\mathcal{M},k}$,
- $(q, R) \rightarrow (q', R')$ if and only if there exist $T, T' \in m_{k,\mathcal{M}}(\mathcal{T})$ with $\text{unord}(T) = R, \text{unord}(T') = R'$ such that $(q', f_{T'}, m_{T'})$ is a valid successor configuration from (q, f_T, m_T) ,
- $(q, T) \leq (q', T')$ if and only if $q = q'$ and $T \leq_C T'$ as in Definition 2.1.12, where $(A, \leq) = (\mathcal{M} \cup \{\#\}, =)$.

It is clear that this is a WSTS, using the same reasoning as for $\mathcal{S}_{\mathcal{A}}$. Moreover, $(q_0, R_0) \rightarrow (q_1, R_1) \rightarrow \dots \rightarrow (q_n, R_n)$ in $\mathcal{S}'_{\mathcal{A}}$, if and only if $(q_0, T_0) \rightarrow (q_1, T_1) \rightarrow \dots \rightarrow (q_n, T_n)$ in $\mathcal{S}_{\mathcal{A}}$ for some $T_0, T_1, \dots, T_n \in m_{k,\mathcal{M}}(\mathcal{T})$, where $R_i = \text{unord}(T_i)$. In fact, we could have used this WSTS for our previous results. We used the graphs from $m_{k,\mathcal{M}}(\mathcal{T})$ directly as this simplifies the notation.

Hence, termination for $\mathcal{S}_{\mathcal{A}}$ is decidable if and only if termination for $\mathcal{S}'_{\mathcal{A}}$ is decidable. We only have left to show that $\mathcal{S}'_{\mathcal{A}}$ is image-finite and post-effective. This WSTS is image-finite as given a memory graph configuration, we can only relabel the nodes, and add at most one new node. There are finitely many ways to do this, hence only finitely many configurations in $\text{Post}(s)$. Moreover, we can calculate these effectively by iterating through every possible application of every possible transition of $\mathcal{A}$. $\square$

Remark 4.2.8. Theorem 4.2.7 does not seem to be immediately applicable to our theory of r-NDCMA, but we mention it here as it may be useful for future applications. One possible way we could apply this result is if we extended these NDCMA to accept ω -words. If every state of $\mathcal{A}$ was accepting, then $\mathcal{A}$ would be empty if and only if $\mathcal{S}_{\mathcal{A}}$ terminates given the start configuration $(q_0, T_{\#})$. It might also be interesting to investigate whether this result would imply that it is decidable whether for a given r-NDCMA $\mathcal{A}$, the length of any $w \in \mathcal{L}(\mathcal{A})$ is bounded.

4.3 Closure Properties

Definition 4.3.1. A k -rigid ϵ -NDCMA is a k -rigid NDCMA, additionally equipped with set of ϵ -transitions $\delta_{\epsilon} : Q \rightarrow \mathcal{P}(Q)$ which can be fired without reading any

input. We denote these transitions as $q \xrightarrow{\epsilon} q'$ and for any configuration (q, f, m) , if $q \xrightarrow{\epsilon} q'$, then (q', f, m) is a valid successor configuration of (q, f, m) .

Lemma 4.3.2. *Adding ϵ -transitions to r -NDCMA does not increase expressivity.*

Proof. This follows using the typical construction with ϵ -closures. For each state $q \in Q$, we let $\text{eClose}(q) = \{q' : q \xrightarrow{\epsilon^*} q'\}$, and let the new state space be made up of the closures, $Q' = \{\text{eClose}(q) : q \in Q\}$. Then whenever $q' \in \text{eClose}(q)$, $q' \xrightarrow{t} p$, we let $\text{eClose}(q) \xrightarrow{t} \text{eClose}(p)$, and let $\text{eClose}(q)$ be accepting whenever there exists $q' \in \text{eClose}(q)$ which is accepting. We keep everything else in the automaton unchanged, and observe that this new automaton will accept the same language. $\square$

We note that this construction preserves the property that $\mathcal{M}_L = \mathcal{M}$, hence if the ϵ -NDCMA is weak so is the translation. As we see below, this implies that this class of automata is (effectively) closed under union. However this construction is inherently non-deterministic, and we will see later that the result does not hold for deterministic NDCMA.

Lemma 4.3.3. *(Weak) r -NDCMA are effectively closed under union.*

Proof. Let $\mathcal{A} = \langle Q^A, \Sigma, \mathcal{M}, \delta^A, \sigma^A, q_0^A, F_G^A \rangle$, $\mathcal{B} = \langle Q^B, \Sigma, \mathcal{M}, \delta^B, \sigma^B, q_0^B, F_G^B \rangle$ be two k -rigid NDCMA. Note if the memory size k differs for these automata, we can take the maximum value, as every k -rigid automaton can be seen as a k' -rigid automaton for $k' > k$. Then, we can place the two automata next to each other, adding a new start state q_0 with ϵ transitions to q_0^A and q_0^B . This results in a k -rigid NDCMA by Lemma 4.3.2, which accepts the union of the languages. $\square$

An interesting side effect of our definition of r -NDCMA is that the languages they accept are limited because of the way we introduce new data values. Whenever we introduce a new data value, it must be the case that it is a root, or else that its parent is already initialised. For this reason, we define *admissible languages*, which capture this behaviour.

Definition 4.3.4. Given a nested data set $\mathcal{D} = \langle D, \text{pred} \rangle$, a word $w \in (\Sigma \times D)^*$ is said to be *admissible* if $w[i] = (a, d)$ implies that either $\text{pred}(d)$ is undefined,

meaning d is a root, or $\text{pred}(d) = d'$, and $w[j] = (a', d')$ for some $a' \in \Sigma$, $j < i$. Moreover, we say a language $L \subseteq (\Sigma \times D)^*$ is *admissible* if each word $w \in L$ is admissible. We denote the set of all admissible languages over finite alphabet Σ and nested data set $\mathcal{D}$ as $\text{Admiss}(\Sigma, \mathcal{D})$.

Lemma 4.3.5. *Given an r-NDCMA $\mathcal{A}$, $\mathcal{L}(\mathcal{A})$ is admissible.*

Proof. This follows immediately from the fact that by definition of an r-NDCMA, if we see a new data value d then it is either a root, or its parent $\text{pred}(d)$ is initialised. $\square$

Because of this fact, it is clear that the usual complement $(\Sigma \times D)^* / \mathcal{L}(\mathcal{A})$ of the language of an r-NDCMA can never be accepted by an r-NDCMA, as the complement of an admissible language is clearly not admissible. For this reason, we must consider a more constricted definition of complement for r-NDCMAs.

Definition 4.3.6 (Complement). If $\mathcal{L} \in \text{Admiss}(\Sigma, \mathcal{D})$ is an admissible language, we define the complement of $\mathcal{L}$, $\mathcal{L}^c$ to be $\text{Admiss}(\Sigma, \mathcal{D}) / \mathcal{L}$.

Lemma 4.3.7. *Deterministic weak r-NDCMA are effectively closed under complement.*

Proof. Suppose $\mathcal{A} = \langle Q, \Sigma, \mathcal{M}, \delta, \sigma, q_0, F_G \rangle$ is a deterministic weak k -rigid NDCMA. We claim that the automata $\mathcal{A}^c = \langle Q, \Sigma, \mathcal{M}, \delta, \sigma, q_0, Q / F_G \rangle$ obtained by flipping the accepting states accepts $\mathcal{L}(\mathcal{A})^c$. First note that for every $w \in \text{Admiss}(\Sigma, \mathcal{D})$, there exists one and only one run of w in $\mathcal{A}$ since $\mathcal{A}$ is deterministic. Now this run ends with an accepting state in $\mathcal{A}$ if and only if it does not end with an accepting state in $\mathcal{A}^c$, hence $w \in \mathcal{L}(\mathcal{A})$ if and only if $w \notin \mathcal{L}(\mathcal{A}^c)$. $\square$

We prove that the problem of deciding whether the intersection of two arbitrary r-NDCMA is empty is in fact undecidable. We do this by showing that for any Turing machine M we can find NDCMA $\mathcal{A}, \mathcal{B}$ such that $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B}) \neq \emptyset$ if and only if M accepts given the empty tape ϵ . For this we first define a model of Turing machines.

Definition 4.3.8. A Turing machine is a tuple $M = \langle Q, \Sigma, \delta, q_0, F, B \rangle$ where

- Q is a finite set of states,
- Σ is a finite alphabet,
- $\delta : (Q \times \Sigma) \rightarrow Q \times (\Sigma \cup \{\mathbf{L}, \mathbf{R}\})$,
- $q_0 \in Q$ is the start state,
- $F \subseteq Q$ is a set of final states and
- $B \in \Sigma$ is a designated blank symbol.

Theorem 4.3.9. *The problem of determining if $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B}) = \emptyset$ for arbitrary deterministic weak r -NDCMA $\mathcal{A}, \mathcal{B}$ is undecidable.*

Proof. To prove this, we suppose non-emptiness of intersection of two NDCMA is decidable and prove we can decide given an arbitrary TM M whether M accepts given the empty word ϵ . The idea is to encode the contents $v_0, v_1, v_2, \dots$ of the tape in data values $d_0, d_1, d_2, \dots$ such that $d_i = \text{pred}(d_{i+1})$. Since the number of data values allowed is potentially infinite, we can easily do this. Then, we simulate executing the TM by marking one of the values as the head, and moving this pointer according to the moves made, updating the values accordingly.

Since our NDCMA have finite memory, we are not able to access the ‘tape’ freely using only one NDCMA. In particular, the difficulty arises in moving the head to the left or right, as since the memory of a single node is finite, we cannot move back further than its memory. This is the reason why emptiness of a weak NDCMA remains decidable. We can circumvent this with two NDCMA, by defining one in which a node has access to its parent at odd parities, and one in which a node has access to its parent at even parities. This configuration can be seen graphically in Figure 4.1, where $d_i = \text{pred}(d_{i+1})$. This result in fact holds even only using 1-rigid NDCMAs.

Suppose $M = \langle Q, \Sigma, \delta, q_0, F, B \rangle$ is some fixed TM. We represent the transitions in δ graphically as $t_i : (q, a) \rightarrow (p, s)$ where $s \in \Sigma \cup \{\mathbf{L}, \mathbf{R}\}$, and enumerate these as $t_1, t_2, \dots, t_n$. Let $\mathcal{M} = \Sigma \times \{1, 2\} \times \{Head, \overline{Head}\} \times \{HasChild, NoChild\}$, where 1, 2 will be used to denote the parity of a data value, $Head$ will be used to represent that the head of the TM is on the data value’s corresponding cell ($\overline{Head}$ being the

negation of this), *HasChild* and *NoChild* are used to determine whether a node has already spawned a child or not and Σ represents the contents of the tape cell.

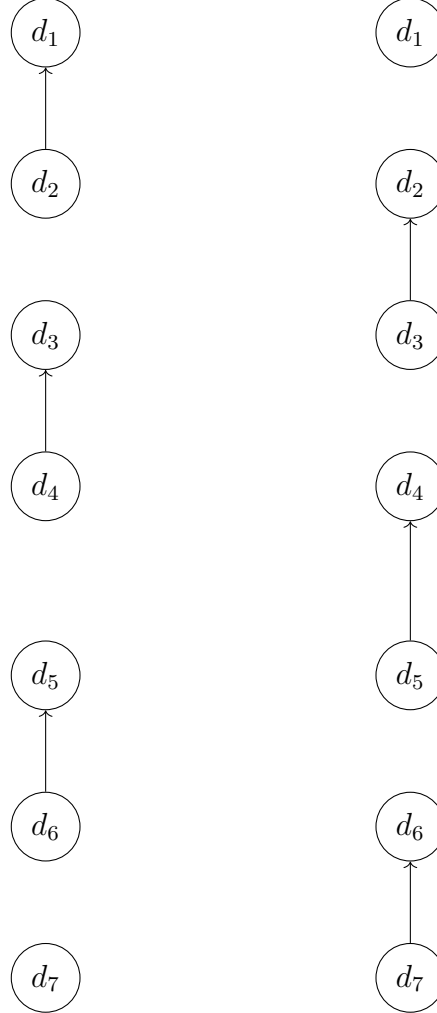


Figure 4.1: Memory graphs for $\mathcal{A}_1$ and $\mathcal{A}_2$ with 7 values.

We define 1-rigid NDCMA $\mathcal{A}_1 = \langle Q', \{1\}, \mathcal{M}, \delta^{(1)}, \sigma^{(1)}, q'_0, F \rangle$ and $\mathcal{A}_2 = \langle Q', \{1\}, \mathcal{M}, \delta^{(2)}, \sigma^{(2)}, q'_0, F \rangle$ on a nested data set $\mathcal{D}$ and singleton alphabet $\{1\}$ with a shared memory set $\mathcal{M}$. These automata are defined to have state space $Q' = Q \cup \{q'_0, t_1^1, \dots, t_n^1, t_1^2, \dots, t_n^2\}$, where Q is the state space of M , the t_i s correspond to transitions of M , and q'_0 is the initial state. Here $\delta^{(1)}$ and $\delta^{(2)}$ will be represented as (partial) functions, which can clearly be made total by adding some sink state $q_\perp$ for all undefined transitions. This means that these automata can be made

deterministic, as required.

These automata have the exact same transitions, except for the initialisation, which places the root node at either parity 1 or 2. $\mathcal{A}_1$ initialises the root to have parity 1 and $\mathcal{A}_2$ initialises the root to have parity 2. Note that we omit the letter $1 \in \Sigma'$ in the transitions as since the alphabet is a singleton, this is the only possible letter. We define the initialisation of the first node as

$$\begin{aligned}\sigma_{(-1)}^1(q'_0, \epsilon) &= (q_0, (B, 1, \textit{Head}, \textit{NoChild}), 0), \\ \sigma_{(-1)}^2(q'_0, \epsilon) &= (q_0, (B, 2, \textit{Head}, \textit{NoChild}), 0).\end{aligned}$$

For the rest of the transitions $\sigma^1 = \sigma^2$ and $\delta^1 = \delta^2$ so we omit the superscript. The idea behind this is when a node has parity 1, it has no memory, and when it has parity 2 it has access to its parent. We use variable names $w \in \mathcal{M}$ and $x \in \Sigma$ and $u \in \{\textit{HasChild}, \textit{NoChild}\}$ when these are arbitrary. Note we use the convention that the same variable used twice means it must remain unaltered. The easiest of these transitions is the case when we write a value, that is when $t_i = (q, a) \rightarrow (p, b)$ where $b \in \Sigma$. For this case we add the transitions

$$\begin{aligned}\delta_0(q, ((a, 1, \textit{Head}, u))) &= (p, ((b, 1, \textit{Head}, u))) \\ \delta_1(q, ((a, 2, \textit{Head}, u), w)) &= (p, ((b, 2, \textit{Head}, u), w))\end{aligned}$$

We need to also take care of the special case when we have an even node that does not have a parent. This does not match the above pattern as our pattern specifies the parent w exists. For this, we add the transition

$$\delta_0(q, ((a, 2, \textit{Head}, u))) = (p, ((b, 2, \textit{Head}, u))).$$

We only need such a transition for $\mathcal{A}_2$, since the root of $\mathcal{A}_2$ is even. These edge cases will appear often in our construction. To aid legibility, we will omit such a transitions, as it is obvious when they are needed.

For the left and right shifts we need to be a bit more careful, as we note that in one automata we will know which data value our new head should point to, and in the other we do not. Moreover, since we are finding the intersection, we want a way to synchronise these automata. For this reason, we first scan the data value corresponding to the head, then the data value corresponding to the new

location of the head. In one of these automata, we have to guess this new location. This also means our transitions are done in two steps, which is why we have the intermediate states t_i^1 and t_i^2 in Q' .

Consider the transition $t_i = (q, a) \rightarrow (p, \mathbf{L})$. There are two possible cases for this — either the head is on a data value of parity 1 or of parity 2. When it is 1, we first scan the head to be able to move it, then pick any arbitrary even value to be the head as follows:

$$\begin{aligned}\delta_0(q, ((a, 1, \text{Head}, u))) &= (t_i^1, ((a, 1, \overline{\text{Head}}, u))), \\ \delta_1(t_i^1, ((x, 2, \overline{\text{Head}}, \text{HasChild}), w)) &= (p, ((x, 2, \text{Head}, \text{HasChild}), w)).\end{aligned}$$

For the case with parity 2, we first scan the head and move the pointer, then scan the new head, making no changes, to synchronise with the other transition. Note that in this case it must be that the node at which the head is first pointing at is not the root, as otherwise our TM would have crashed.

$$\begin{aligned}\delta_1(q, ((a, 2, \text{Head}, u_1), (x, 1, \overline{\text{Head}}, u_2))) &= (t_i^2, ((a, 2, \overline{\text{Head}}, u_1), (x, 1, \text{Head}, u_2))), \\ \delta_0(t_i^1, ((x, 1, \text{Head}, u))) &= (p, ((x, 1, \text{Head}, u))).\end{aligned}$$

The case for the right shift is similar, except for additional σ transitions that handle the initialisation of more tape if it is accessed. In this case the value of $u \in \{\text{HasChild}, \text{NoChild}\}$ is relevant to the transitions, as we ensure a node only spawns one child by only spawning a child when the parent has value *NoChild*, and updating the parent's value to *HasChild* afterwards. Let $t_i = (q, a) \rightarrow (p, \mathbf{R})$. For the odd parity, we first scan the head then the new head, as in the following transitions.

$$\begin{aligned}\delta_0(q, ((a, 1, \text{Head}, u))) &= (t_i^1, (a, 1, \text{Head}, u)), \\ \delta_1(t_i^1, ((x, 2, \overline{\text{Head}}, u_1), (a, 1, \text{Head}, u_2))) &= (p, ((x, 2, \text{Head}, u_1), (a, 1, \overline{\text{Head}}, u_2)))\end{aligned}$$

We handle the case for creating a new node as well, as stated in the transition below.

$$\sigma_0(t_i^1, ((a, 1, \text{Head}, \text{NoChild}))) = (p, ((B, 2, \text{Head}, \text{NoChild}), (a, 1, \overline{\text{Head}}, \text{HasChild}), 1)).$$

For the even parity we first scan the head and update it to reflect that it is no longer the head, then we pick any odd node to set as the new head, as follows:

$$\begin{aligned}\delta_1(q, ((a, 2, \text{Head}, u))) &= (t_i^2, (a, 2, \overline{\text{Head}}, u)), \\ \delta_0(t_i^2, ((x, 1, \overline{\text{Head}}, u))) &= (p, ((x, 1, \text{Head}, u)))\end{aligned}$$

The handling of a new node is as before, updating the label $u \in \{\text{HasChild}, \text{NoChild}\}$.

$$\begin{aligned}\sigma_1(t_i^2, ((a, 2, \text{Head}, \text{NoChild}), w)) &= (p, ((B, 1, \text{Head}, \text{NoChild}), \\ &\quad (a, 2, \overline{\text{Head}}, \text{HasChild}), w), 0).\end{aligned}$$

Note that our transitions reflect that even nodes remember their parents, while odd nodes forget any memory. We also remark that by adding a sink state, we can formulate $\mathcal{A}_1$ and $\mathcal{A}_2$ as deterministic automata as in our definition.

Now, suppose there is some word $w \in \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$. We can write w as a sequence of data values, $d_1, d_2, d_3, \dots$, which are accepted by both $\mathcal{A}_1$ and $\mathcal{A}_2$. Now, $\mathcal{A}_1$ ensures that left and right shifts are correct for even and odd locations of the head respectively, meanwhile $\mathcal{A}_2$ ensures that left and right shift are correct for odd and even locations of the head respectively, hence when we take the intersection, $d_1, d_2, \dots$ represents the locations of the head for a valid run of the TM, which accepts ϵ . Therefore, it must be the case that M accepts given ϵ . Moreover, if $\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2) = \emptyset$, then there is no possible sequence of data values $d_1, d_2, d_3, \dots$ that represent the location of the head in an accepting run of the TM, M , so M does not accept given ϵ . $\square$

Remark 4.3.10. This construction can also be used to show that an NDCMA in which each node d has access to just its parent has undecidable emptiness, as in fact the simulation only depends on the fact that each node can access and modify the label of its parent.

Corollary 4.3.11. *Let $\mathcal{A}, \mathcal{B}$ be weak deterministic k -rigid NDCMA. Then, we cannot always effectively construct an NDCMA $\mathcal{C}$ such that $\mathcal{L}(\mathcal{C}) = \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$. That is, weak r -NDCMA are not effectively closed under intersection, not even if the initial NDCMA were deterministic.*

Proof. This follows immediately from the fact that if we could construct a weak NDCMA $\mathcal{C}$ such that $\mathcal{L}(\mathcal{C}) = \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$, then because of Corollary 4.2.6, we can decide whether this intersection is empty, contradicting Theorem 4.3.9. $\square$

Lemma 4.3.12. *Deterministic weak r -NDCMA are not effectively closed under union.*

Proof. Suppose they are effectively closed under union. Then, given arbitrary deterministic k -rigid NDCMA $\mathcal{A}, \mathcal{B}$, we can construct deterministic $\mathcal{A}^c, \mathcal{B}^c$. Then by our assumption, there exists a deterministic $\mathcal{C}$ which accepts $\mathcal{L}(\mathcal{A}^c) \cup \mathcal{L}(\mathcal{B}^c)$. Since this automaton is deterministic, we can further construct its complement $\mathcal{C}^c$ which accepts $(\mathcal{L}(\mathcal{A}^c) \cup \mathcal{L}(\mathcal{B}^c))^c = \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$, contradicting the fact that emptiness of intersection of two r -NDCMAs is undecidable. $\square$

Definition 4.3.13. We say that a language $\mathcal{L} \subseteq (\Sigma \times D)^*$ is a *trace language* if whenever $uw \in \mathcal{L}$ for $u, w \in (\Sigma \times D)^*$, $u \in \mathcal{L}$. Moreover, an automaton $\mathcal{A}$ is a *trace-accepting* automata whenever $\mathcal{L}(\mathcal{A})$ is a trace language.

Note that a weak automata $\mathcal{A}$ is a trace-accepting automata if and only if there exists an equivalent automata $\mathcal{A}'$ with only accepting states. We use the above definition instead of this one, as otherwise we would not be able to adequately define deterministic trace-accepting automata, as a deterministic automaton with only accepting states accepts all admissible words.

Lemma 4.3.14. *A language $L \in \text{Admiss}(\Sigma, \mathcal{D})$ is a trace language if and only if for $u, w \in (\Sigma \times D)^*$ whenever $u \in L^c$ and uw is an admissible word, then $uw \in L^c$.*

Proof. L is admissible if and only if for all $uw \in L$, $u \in L$. Taking the contrapositive, this is equivalent to saying $u \notin L$ implies $uw \notin L$, which is furthermore equivalent to saying $u \in L^c$ implies $uw \in L^c$ for each uw admissible. $\square$

Theorem 4.3.15. *Suppose $\mathcal{A}, \mathcal{B}$ are trace-accepting deterministic weak r -NDCMA. Then, the problem of determining if $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})^c = \emptyset$ is undecidable.*

Proof. This follows by the same construction as in Theorem 4.3.9. We set $\mathcal{A}$ as constructed in this theorem to be a trace-accepting language by making all the states accepting, then adding a sink state to make the automata deterministic as

before. Now, we construct $\mathcal{B}'$ as in the previous theorem, except that once we reach an accepting state, we further add all possible transitions as loops, ensuring that if a word u is accepted by $\mathcal{B}'$, then any admissible extension uw of it is also accepted. Note we can easily make $\mathcal{B}'$ deterministic. By Lemma 4.3.14, $\mathcal{L}(\mathcal{B}') = \mathcal{L}(\mathcal{B})^c$ for some trace-accepting weak deterministic 1-rigid NDCMA $\mathcal{B}$, and hence our theorem follows. $\square$

Corollary 4.3.16. *Given two trace-accepting deterministic weak r -NDCMA $\mathcal{A}, \mathcal{B}$, it is undecidable whether $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$*

Proof. This follows immediately from Theorem 4.3.15, and the fact that $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ holds if and only if $\mathcal{L}(\mathcal{A}) \cap (\mathcal{L}(\mathcal{B}))^c = \emptyset$. $\square$

Corollary 4.3.17. *Given two (trace-accepting) weak r -NDCMA $\mathcal{A}, \mathcal{B}$, it is undecidable whether $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$*

Proof. Suppose language equivalence was decidable for trace-accepting languages. Then, given arbitrary deterministic weak k -rigid NDCMA $\mathcal{A}, \mathcal{B}$, we can construct a weak k -rigid NDCMA $\mathcal{C}$ accepting $\mathcal{L}(\mathcal{A}) \cup \mathcal{L}(\mathcal{B})$ by Lemma 4.3.3. Note that the union of two trace-accepting languages is clearly also a trace-accepting language. Then, we can decide if $\mathcal{L}(\mathcal{C}) = \mathcal{L}(\mathcal{B})$ which is true if and only if $\mathcal{L}(\mathcal{A}) \cup \mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{B})$ which is equivalent to $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$. However, this is undecidable by Corollary 4.3.16. $\square$

Remark 4.3.18. While we have obtained various undecidability results for our r -NDCMA, we have not yet managed to obtain any positive or negative results for deciding equivalence of *deterministic* weak r -NDCMA. Inclusion for this class of automata has been proven to be undecidable, but this does not imply that equivalence is necessarily undecidable. In fact, there is a model of automata — deterministic pushdown automata, for which the inclusion problem is undecidable, but equivalence is decidable [37]. This means there is still some hope for our model to have decidable equivalence.

While we do not have an answer to equivalence for deterministic weak r -NDCMA, we do have a negative result for equivalence of the Σ -projection language

of these automata, which is the ‘finite part’ of the language. We present this result in this section, but leave the proof for Section 4.4, as this result relies on an undecidability result on reset Petri nets which we will prove.

We also note that this result is true for the subclass of bounded NDCMA, which coincide with the NDCMA model in [14]. This highlights the fact that this result cannot be applied to show that equivalence of deterministic weak r-NDCMA is undecidable, as in fact equivalence of weak bounded NDCMA is decidable.

Definition 4.3.19. Let $L \subseteq (\Sigma \times D)^*$ be a data languages. The Σ -*projection language* of $\text{Proj}(L)$ is the set $\text{Proj}(L) = \{a_1 a_2 \dots a_n \in \Sigma^* \mid \exists d_1, d_2, \dots d_n \in D \text{ such that } (a_1, d_1)(a_2, d_2) \dots (a_n, d_n) \in L\}$.

Theorem 4.3.20. For $k \geq 1$, given two deterministic trace k -rigid bounded NDCMA $\mathcal{A}, \mathcal{B}$ it is undecidable whether $\text{Proj}(\mathcal{L}(\mathcal{A})) = \text{Proj}(\mathcal{L}(\mathcal{B}))$.

4.4 Petri Nets and Simulation

The purpose of this section is twofold. Firstly, to obtain a proof of Theorem 4.3.20 we make use of a new undecidability result for reset Petri nets, hence to do this we introduce Petri nets and how they work. Moreover, in the process of building up the theory of Petri nets, we will briefly summarise their closure results and compare these with the ones we have obtained for r-NDCMA. This allows us to compare our new model of automata with a well-known model of computation.

Petri nets are made up of *places*, which can have any number of tokens, and *transitions* which along with a set of *arcs* specify how we are able to move tokens. We call a configuration of tokens of a Petri net a *marking*, which is a function from the set of places to the natural numbers. Formally, we can describe a Petri net as below.

Definition 4.4.1. A Petri net $N = \langle \Pi, T, \sigma, E, M_0 \rangle$ is made up of

- A finite set of *places*, $\Pi = \{r_1, r_2, \dots, r_{|\Pi|}\}$,
- A finite set of *transitions*, $T = \{t_1, \dots, t_{|T|}\}$,
- A *labelling function* $\sigma : T \rightarrow \Sigma$, where Σ is some finite alphabet,

- A finite set of *arcs* $E \subseteq (\Pi \times T) \cup (T \times \Pi)$ and
- An *initial marking* $M_0 : \Pi \rightarrow \mathbb{N}$.

Given a transition $t \in T$ and place $r \in \Pi$, r is an input place of t if $(r, t) \in E$ and an output place of t if $(t, r) \in E$. Then, given two markings M_1, M_2 and a Petri net $N = \langle \Pi, T, E, M_0 \rangle$, we can go from M_1 to M_2 with transition $t \in T$, denoted $M_1 \xrightarrow{t} M_2$ if:

- Whenever r is an input place of t , $M_1(r) > 0$ and
- M_2 is the marking obtained by subtracting one from each input place of t , and adding one to each output place of t .

If it is possible to use a transition t from a marking M , we say t is fireable from M . Furthermore, we define $M_1 \xrightarrow{s} M_2$ for $s \in T^*$ as

- $M_1 \xrightarrow{ts} M_2$ if there exists M' such that $M_1 \xrightarrow{t} M'$ and $M' \xrightarrow{s} M_2$,
- $M \xrightarrow{\epsilon} M$.

The prefix language of a Petri net, $\mathcal{L}(N)$ is the image under σ of the set of sequences $s \in T^*$ which are fireable from the initial marking M_0 , that is $\mathcal{L}(N) = \{\sigma(s) : \exists M M_0 \xrightarrow{s} M\}$. Given some set F of final markings, we further define $\mathcal{L}_F(N)$ to be the set $\{\sigma(s) : \exists M \in F M_0 \xrightarrow{s} M\}$. These languages are referred to as P-type and L-type languages respectively in the literature. In the case when σ is one to one, we call the Petri net N and its resulting language deterministic.

We can also define the *reachability* and *coverability* decision problems for Petri nets. For coverability, we provide an order on markings, which is the standard component-wise order: $M \leq M'$ if and only if $M(r) \leq M'(r)$ for each $r \in \Pi$.

Definition 4.4.2. Given a set of final markings F and a Petri net $N = \langle \Pi, T, \sigma, E, M_0 \rangle$, we can ask the following questions:

1. **Reachability:** Is there a sequence of transitions s such that $M_0 \xrightarrow{s} M \in F$?
2. **Coverability:** Is there a sequence of transitions s such that $M_0 \xrightarrow{s} M'$ for some marking $M' \geq M \in F$.

We present a table of closure results for Petri net languages, alongside the closure results for weak r-NDCMA. The closure results under union, intersection and complement for Petri nets can be seen in the table below, and are summarised in Chapter 6 of [31]. Undecidability of equivalence/inclusion for both L-type and P-type (even for Petri nets with exactly one unbounded place) was shown in [20].

Closure properties for deterministic L-type languages were shown in [30], from which decidability of inclusion follows by the fact that the complement of a deterministic Petri net language is again a Petri net language. In fact this implies the stronger result of decidability of inclusion $L_1 \subseteq L_2$, where L_1 is an L-type language and L_2 is a deterministic L-type language. This result was originally shown through decidability of reachability, as for deterministic Petri nets inclusion, equivalence and reachability are mutually reducible [1].

	L-type	P-type	det. L-type	r-NDCMA	det. r-NDCMA
Union	✓	✓	X	✓	X
Intersection	✓	✓	✓	X	X
Complement	X	X	L-type	X	✓
Inclusion	X	X	✓	X	X
Equivalence	X	X	✓	X	?

We further study an extension of Petri nets: reset Petri nets. In these nets we additionally have *reset arcs*, which reset the number of tokens in their input places to zero. In this case, our arcs are now partitioned into disjoint sets $E = E_r \cup E_{nr} \cup E_{out}$, where $E_r, E_{nr} \subseteq \Pi \times T$, $E_{out} \subseteq T \times \Pi$. We call r a non-reset input place of t if $(r, t) \in E_{nr}$ and a reset input place of t if $(r, t) \in E_r$. Moreover, we now say $M_1 \xrightarrow{t} M_2$ if:

- Whenever r is a non-reset input place of t , $M_1(r) > 0$ and
- M_2 is the marking obtained by subtracting one token from each non-reset input place of t , removing all tokens from each reset input place of t , and adding one token to each output place of t .

The remaining definitions and notation for Petri nets can now be lifted for reset Petri nets. While coverability is still decidable for these nets, reachability becomes undecidable as was shown in [1], Theorem 5. This is done by reducing the halting

problem of 2-counter machines to reachability of reset Petri nets. We introduce these machines below, as we will use them to prove our undecidability result.

Definition 4.4.3. A 2-counter machine K consists of

- A finite set of states Q and distinguished start and end states $s_0, s_f \in Q$,
- A finite set of increment transitions $T_{\text{inc}} \subseteq S \times S \times \{1, 2\}$ and
- A finite set of decrement/jump transitions $T_{\text{dec}} \subseteq S \times S \times S \times \{1, 2\}$.

A configuration of the machine consists of a tuple $(s, c_1, c_2) \in Q \times \mathbb{N} \times \mathbb{N}$ which contains the current state, and the current values of the counters c_1, c_2 .

Given a transition $t = (s, s', 1) \in T_{\text{inc}}$, we can transition from (s, c_1, c_2) to $(s', c_1 + 1, c_2)$, and similarly we increment the counter c_2 if $t = (s, s', 2)$. If $t = (s, s', s'', 1) \in T_{\text{dec}}$, we can transition from $(s, c_1 + 1, c_2)$ to (s', c_1, c_2) (noting $c_1 \geq 0$, so the counter is non-zero) and from $(s, 0, c_2)$ to $(s'', 0, c_2)$. This means we end up in state s' if the value of the first counter is non-zero, and in state s'' otherwise. The transition $(s, s', s'', 2)$ is defined similarly.

We say that a 2-counter machine K halts if there is a set of transitions from configuration $(s_0, 0, 0)$ to configuration $(s_f, 0, 0)$.

It is well-known that the halting problem for 2-counter machines is undecidable in general. This fact was used to prove that reachability is undecidable for reset Petri nets. We go through the details of this proof, as we adapt the construction to prove further results on the languages of these nets. Note that in this construction the labelling σ is irrelevant so we do not mention it.

Theorem 4.4.4 (Undecidability of reachability [1]). *Let K be a 2-counter machine. There exists a reset Petri net N and a configuration M such that M is reachable from N if and only if K halts. Therefore, reachability is undecidable for reset Petri nets.*

Proof (Sketch). Let K be a 2-counter machine. We construct a Petri net N which essentially simulates the behaviour of K using 4 extra places for the counters (2 for each counter) C_1, C'_1, C_2, C'_2 . The main challenge of this simulation is modelling the decrement/jump transition. To tackle this, we create two transitions, one

which decrements the counter, and another which assumes the counter is zero. In the case where the counter is assumed to be zero, we keep C_i unchanged, but we reset the value of C'_i . This way, we essentially make it so the counters are in sync if and only if each zero transition is faithful to the original 2-counter machine.

We formalise this idea further below. Firstly, if K has the states $s_1, \dots, s_n$, we set the places of N to be $S_1, \dots, S_n$ corresponding to the states, with the additional places C_1, C_2, C'_1, C'_2 to model the counters. We use S_0, S_f to denote the corresponding places to s_0, s_f . These places will be used to model the current state, by placing a singular token in the place corresponding to current state, and having the rest be empty.

For each $t = (s, s', i) \in T_{\text{inc}}$ we add a transition $t \in T$, and arcs $(S, t) \in E_{\text{nr}}$, $(t, S'), (t, C_i), (t, C'_i) \in E_{\text{out}}$. This handles incrementing the counter c_i and the change of state. For each $t = (s, s', s'', i) \in T_{\text{dec}}$, we add two transitions, t_Z , corresponding to the assumption that c_i has value zero, and t_{NZ} corresponding to the assumption that c_i is non-zero. For t_Z , we add the arcs $(S, t_Z) \in E_{\text{nr}}$, $(t_Z, S'') \in E_{\text{out}}$ as well as the reset arc $(C'_i, t_Z) \in E_r$. This resets the value of C'_i in the case it is not zero. For t_{NZ} , we add the arcs $(S, t_{\text{NZ}}), (C_i, t_{\text{NZ}}), (C'_i, t_{\text{NZ}}) \in E_{\text{nr}}$ and $(t_{\text{NZ}}, S') \in E_{\text{out}}$.

It is clear from this construction that the counter C'_i will always contain a lower value to C_i , since it is incremented/decremented similarly, and it is sometimes reset, so can have a lower value. It is also clear that the two counters are equal if and only if each decrement/jump operation was executed faithfully, hence the Petri net has faithfully simulated the machine K . Moreover, it can be deduced that if $C_i = 0$, then $C'_i = 0$, hence the marking with $M(S_f) = 1$, $M(r) = 0$ otherwise is reachable if and only if K halts. $\square$

We further adapt the construction from [1] which was used to show that reachability is reducible to equivalence for deterministic Petri nets. This construction works by keeping track of the sum of all places, hence having a transition that is fireable if and only if every place has zero tokens. Our goal is to reduce reachability of reset nets to equivalence of reset nets. However, this would be hard to do in general as it is unclear how to simulate resets when we are keeping the sum of all places. We circumvent this issue by working only on the N constructed above.

This is easier as we can focus only on C_1 and C_2 , as these are never reset, and them being zero implies C'_1 and C'_2 are zero. Combining these two ideas, we obtain a new undecidability result for reset nets.

Corollary 4.4.5. *Language equivalence of deterministic prefix reset Petri net languages is undecidable.*

Proof. Let K be a 2-counter machine, and let N be the Petri net constructed as above such that the configuration M with $M(S_f) = 1, M(r) = 0$ otherwise is reachable if and only if K halts. We construct deterministic nets N_1, N_2 such that $\mathcal{L}(N_1) \neq \mathcal{L}(N_2)$ if and only if M is reachable from N .

These nets have identical transitions to N except for an additional transition t_0 which we will use to separate the two languages. Since these are deterministic nets, we can assume that the labelling function σ is just the identity and ignore it from now on.

We let the places of N_1, N_2 be the places of N , with an additional new place C . To N_2 , we add an additional place D , which will keep track of the sum. To both N_1, N_2 , we add the arcs $(C, t_0), (S_f, t_0) \in E_{\text{nr}}$ and the arc (t_0, S_f) . In the net N_1 , this transition will always be fireable when we are in state s_f . We will make it such that in N_2 , the transition is fireable if and only if we are in state s_f and the marking M is not yet reached.

Firstly, we add the arcs $(D, t_0) \in E_{\text{nr}}, (t_0, D) \in E_{\text{out}}$ to the net N_2 . This means t_0 is fireable if and only if D is non-empty, and firing t_0 does not change the contents of D . Now, for each transition t which increments C_1, C_2 (these correspond to $t \in T_{\text{inc}}$), we add the arc $(t, D) \in E_{\text{out}}$, which means the counter D is also incremented, and for each transition t which decrements C_1, C_2 (these correspond to the transitions t_{NZ} for $t \in T_{\text{dec}}$), we add the arc $(D, t) \in E_{\text{nr}}$ meaning D is also decremented. This means D keeps track of the sum $C_1 + C_2$, and hence t_0 is fireable if and only if $C_1 + C_2 > 0$ and we are in the final state S_f .

If it is always the case that $C_1 + C_2 = 0$ when we are in the final state, then N_1 and N_2 accept the exact same language as for both these languages t_0 is fireable if and only if there is a token in place S_f . However, if at any point $C_1 + C_2 = 0$ and there is a token in S_f , then t_0 is fireable in N_1 but not in N_2 , meaning their languages are not equal. Because of the construction of C_1, C_2 in N , $C_1 + C_2 = 0$ if

and only if $C_1 = C'_1 = C_2 = C'_2 = 0$ hence this situation happens if and only if K halts. Therefore, if we can decide language equivalence, we can decide the halting problem for 2-counter machines, which concludes our proof of undecidability of equivalence. $\square$

Now, in [12], the author is able to simulate reset Petri nets using bounded NDCMA. We make use of the underlying ideas behind this construction, noting again that bounded NDCMA are a subclass of r-NDCMA. At initial glance this simulation seems incompatible with the above undecidability result, as in fact equivalence for deterministic bounded depth NDCMA is decidable. However the construction explicitly spells out which places are being incremented/decremented in the data values, meaning we can't faithfully compare deterministic reset net Petri languages by directly comparing r-NDCMA languages. While this undecidability result cannot be lifted directly to r-NDCMA we can in fact lift it to the 'projection language' of the automata, meaning the languages obtained by only considering the finite alphabet part of the execution.

Another consideration we need to handle is in the simulation in [12] increments and decrements are done one step at a time. This means to simulate one transition our automata takes multiple steps. To account for this, we have to modify the simulation so that each transition takes the same number of steps, in particular the maximum number of steps needed. We must also be sure to have the same initialisation step in both automata. We present the simulation below.

Theorem 4.4.6. *If it is decidable given two deterministic trace k -rigid bounded NDCMA $\mathcal{A}, \mathcal{B}$ whether $\text{Proj}(\mathcal{L}(\mathcal{A})) = \text{Proj}(\mathcal{L}(\mathcal{B}))$, then we can decide whether $\mathcal{L}(N_1) = \mathcal{L}(N_2)$ for arbitrary deterministic reset nets N_1, N_2 .*

Proof. Let $N_1 = \langle \Pi_1, T, \text{id}, E_1, M_{0,1} \rangle$, $N_2 = \langle \Pi_2, T, \text{id}, E_2, M_{0,2} \rangle$ be arbitrary reset nets over the same transitions and the identity labelling function. We construct two automata simulating N_1, N_2 respectively such that their projection language is equal if and only if $\mathcal{L}(N_1) = \mathcal{L}(N_2)$. Our alphabet will be $\Sigma = T \cup \{\text{init}\}$, where *init* will denote that we are initialising the Petri net. We initialise some variables to handle initialisation of places, initialisation of tokens and repetition of actions in the case where a transition needs multiple increments/decrements.

- Let p be the maximum of $|\Pi_1|, |\Pi_2|$. We will enumerate the places in Π_1, Π_2 as $\Pi_1 = \{r_{1,1}, r_{2,1}, \dots, r_{p,1}\}$, $\Pi_2 = \{r_{1,2}, r_{2,2}, \dots, r_{p,2}\}$, noting that we can add extra places without affecting the accepted language.
- Let m be the maximum number of tokens in $M_{0,1}$ and $M_{0,2}$.
- Let w be the maximum number of steps needed for a transition. Increments and decrements require one step, while resets require two, so given t with k_1 non-reset input places, k_2 reset input places and k_3 output places, w is the maximum of $k_1 + 2k_2 + k_3$ over all t for $E \in \{E_1, E_2\}$.

The projection words our automata will accept will be prefixes of words with $p + m + 1$ *init* letters, followed by letters $t \in T$ where each t is repeated w times.

Our automata will start the same, by initialising $p + 1$ level 1 data values with the labels $1, 2, \dots, p, p + 1$. This is done in $p + 1$ transitions, and is permitted with the label *init* $\in \Sigma$. The data value with label $p + 1$ will act as a dummy for when the automata does not need to do any further actions, but has not done a total of w actions for a transition yet.

Now, for m steps we will add the tokens to our places, by adding children to our data values. From each marking we can obtain a list $i_1, i_2, \dots, i_m$, where i_j is an index of a place such that if a place i_j has some number u of tokens, i_j appears u times in the list. We use the label $+$, $-$ to denote an active token and a decremented token, and hence adding these tokens can be modelled with states $p_1, \dots, p_m$, and the transition $p_j \xrightarrow[(+, i_j), m=1]{init, (\perp, i_j)} p_{j+1}$. If the Petri net does not have m tokens, we pad the computation by accepting the dummy value with label $p + 1$, with the transition $p_j \xrightarrow[p+1]{init, (p+1)} p_{j+1}$.

After this initialisation, we have $p + 1$ places, where the i th place has as many children as it has tokens in the initial marking. This initialisation also takes the same number of steps, so we can be assured that our two automata still match on their projection language. We can now describe how we simulate transitions, following closely the process in [12].

First note we need some kind of landing state p from which we will start the computation. From this, after reading a value with the finite part indicating t we will move to state p_1^t then p_2^t up to p_{w-1}^t to complete all the steps required for the

transition, then we move back to the landing state p . All the data values read in this sequence must have finite part t , otherwise the automaton crashes. The actions of the automata are carried out as following:

- **Increment:** Increment of a place i is modelled by adding a child to the level 0 value with label i . This is done with the transition $p_j^t \xrightarrow[(+,i),m=1]{t,(\perp,i)} p_{j+1}^t$, where we add a new child to the i node with label $+$, which remembers its parent.
- **Decrement:** Decrement of a place i is modelled by change the label of a positive child of the level 0 value with label i . This is done with the transition $p_j^t \xrightarrow[(-,i)]{t,(+,i)} p_{j+1}^t$, where we change the label of a child of the node labelled with i from $+$ to $-$.
- **Reset:** Reset is done in two steps. First we must erase the level 0 value with the label i , by matching with label i and updating the label to $-$, using the transition $p_j^t \xrightarrow[(-)]{t,(i)} p_{j+1}^t$. Then we create a new level 1 data value with the label i , using the transition $p_{j+1}^t \xrightarrow[(i),m=0]{t,(\perp)} p_{j+2}^t$. This new data value will have no children hence has no tokens ascribed to it.

If the maximum number of steps is not needed, we simply have some dummy transition on the dummy place $p + 1$. This way, each time a data value with label $t \in \Sigma$ is read, the tokens of net are updated accordingly.

Now the automata constructed will accept as a projection language prefixes of words of the form $(\text{init})^{p+m+1}(t_1)^w(t_2)^w \dots (t_n)^w$ where $t_1 t_2 \dots t_n$ is a valid word in the corresponding reset Petri net. Therefore it is clear that these two languages are equal if and only if the prefix languages of the reset Petri nets are equal. We note that in this construction, the memory of a child data value was always of size 1, meaning it was maximal. This means that the NDCMA constructed are in fact bounded NDCMA and correspond to those defined in [14] — hence our result applies to the nested data automata defined in this paper as well. $\square$

Combining the results of Theorem 4.4.6 and Corollary 4.4.5, we can conclude Theorem 4.3.20 as desired.

Chapter 5

Applying Unbounded NDCMA

In this chapter, we will show the expressive power of r-NDCMA by representing programming language traces with these automata. In particular, we will study the trace language of FOSC in HOSC contexts, which is a fragment of the programming language ML. Throughout this chapter, we will use NDCMA to refer to weak r-NDCMA unless otherwise stated.

5.1 Language Syntax and Semantics

In this section we will establish the syntax and semantics for FOSC in HOSC contexts, which is a fragment of the programming language ML. In particular we will study a version of FOSC as in [9] without recursion and with an additional while constructor used to represent loops. The syntax of HOSC is specified in Figure 5.1, and the operational semantics is specified in Figure 5.2. FOSC is the restriction of HOSC, to references of base types. The types hence are defined as

$$\sigma, \tau \triangleq \beta \mid \text{ref} \beta \mid \tau \times \sigma \mid \tau \rightarrow \sigma \mid \text{cont } \tau$$

for $\beta \in \{\text{Unit}, \text{Int}, \text{Bool}\}$. Note that we also assume these base types to be finite, so in particular there is some maximum integer max .

Trace models for HOSC were established in [21]. Loops were not worked with explicitly in this language. Instead, recursion, which is stronger than loops, was used. Fortunately we can express a while loop in terms of recursion, which implies

$$\begin{aligned}
\sigma, \tau &\triangleq \text{Unit} \mid \text{Int} \mid \text{Bool} \mid \text{ref}\tau \mid \tau \times \sigma \mid \tau \rightarrow \sigma \mid \text{cont } \tau \\
U, V &\triangleq () \mid \mathbf{tt} \mid \mathbf{ff} \mid \widehat{n} \mid x \mid \ell \mid \langle U, V \rangle \mid \lambda x^\tau. M \mid \text{cont}_\tau K \\
M, N &\triangleq V \mid \langle M, N \rangle \mid \pi_i M \mid MN \mid \text{ref}_\tau M \mid !M \mid M := N \mid \text{if } M_1 \ M_2 \ M_3 \\
&\quad \mid M \oplus N \mid M \sqsupset N \mid M = N \mid \text{call/cc}_\tau(x.M) \mid \text{throw}_\tau M \text{ to } N \\
&\quad \mid \text{while } M \text{ do } N \\
K &\triangleq \bullet \mid \langle V, K \rangle \mid \langle K, M \rangle \mid \pi_i K \mid VK \mid KM \mid \text{ref}_\tau K \mid !K \mid V := K \\
&\quad \mid K := M \mid \text{if } K \ M \ N \mid K \oplus M \mid V \oplus K \mid K \sqsupset M \mid V \sqsupset K \mid K = M \\
&\quad \mid V = K \mid \text{throw}_\tau V \text{ to } K \mid \text{throw}_\tau K \text{ to } M \\
C &\triangleq \bullet \mid \langle M, C \rangle \mid \langle C, M \rangle \mid \pi_i C \mid \lambda x^\tau. C \mid MC \mid CM \mid \text{ref}_\tau C \mid !C \mid C := M \\
&\quad \mid M := C \mid \text{if } C \ M \ N \mid \text{if } M \ C \ N \mid \text{if } M \ N \ C \mid C \oplus M \mid M \oplus C \\
&\quad \mid C \sqsupset M \mid M \sqsupset C \mid C = M \mid M = C \mid \text{call/cc}_\tau(x.C) \mid \text{throw}_\tau C \text{ to } M \\
&\quad \mid \text{throw}_\tau M \text{ to } C \mid \text{while } C \text{ do } M \mid \text{while } M \text{ do } C
\end{aligned}$$

Notational conventions: $x, y \in \mathbf{Var}$, $\ell \in \mathbf{Loc}$, $n \in \{0, \dots, \max\}$, $i \in \{1, 2\}$,
 $\oplus \in \{+, -, *\}$, $\sqsupset \in \{=, <\}$
Syntactic sugar: $\text{let } x = M \text{ in } N$ stands for $(\lambda x. N)M$ (if x does not occur in N we also write $M; N$)

Figure 5.1: HOSC syntax

that the trace model is still correct for our language with loops. Recursion is defined using the construct $\mathbf{rec } y(x^\sigma).M$, with reduction semantics as follows:

$$K[\underbrace{(\mathbf{rec } y(x^\sigma).M)V}_U] \rightarrow K[M\{V/x, U/y\}].$$

Using this notation, we can define while using recursion as

$$\text{while } M \text{ do } N \triangleq (\mathbf{rec } y(x^{\text{Unit}}).\text{if } M \ (N; y()) \ ()) ().$$

This term reduces as we hope as

$$\begin{aligned}
K[(\mathbf{rec } y(x^{\text{Unit}}).\text{if } M \ N; y() \ ()) \ ()] &\rightarrow K[\text{if } M \ (N; (\mathbf{rec } y(x^{\text{Unit}}).\text{if } M \ N; y() \ ())) \ ()] \\
&= K[\text{if } M \ (N; \text{while } M \text{ do } N) \ ()].
\end{aligned}$$

Definition 5.1.1. Given a HOSC term $\Gamma \vdash M : \tau$ we refer to the types in Γ and τ as *boundary types*.

$(K[(\lambda x^\sigma.M)V], h)$	$\rightarrow (K[M\{V/x\}], h)$	
$(K[\pi_i \langle V_1, V_2 \rangle], h)$	$\rightarrow (K[V_i], h)$	
$(K[\text{if } \mathbf{tt} \ M_1 \ M_2], h)$	$\rightarrow (K[M_1], h)$	
$(K[\text{if } \mathbf{ff} \ M_1 \ M_2], h)$	$\rightarrow (K[M_2], h)$	
$(K[\widehat{n} \oplus \widehat{m}], h)$	$\rightarrow (K[\widehat{n \oplus m}], h)$	
$(K[\widehat{n} \sqcap \widehat{m}], h)$	$\rightarrow (K[b], h)$	$b = \mathbf{tt}$ if $n \sqcap m$, o/w $b = \mathbf{ff}$
$(K[\text{call/cc}(x^\tau.M)], h)$	$\rightarrow (K[M\{\text{cont}_\tau K/x\}], h)$	
$(K[!\ell], h)$	$\rightarrow (K[h(\ell)], h)$	
$(K[\text{ref } V], h)$	$\rightarrow (K[\ell], h \cdot [\ell \mapsto V])$	
$(K[\ell := V], h)$	$\rightarrow (K[()], h[\ell \mapsto V])$	
$(K[\ell = \ell'], h)$	$\rightarrow (K[b], h)$	$b = \mathbf{tt}$ if $\ell = \ell'$, o/w $b = \mathbf{ff}$
$(K[\text{throw}_\tau V \text{ to } \text{cont}_\tau K'], h)$	$\rightarrow (K'[V], h)$	
$(K[\text{while } M \text{ do } N], h)$	$\rightarrow (K[\text{if } M \ N; \text{while } M \text{ do } N \ ()], h)$	

Figure 5.2: Operational reduction for HOSC

We will only consider ‘cr-free’ terms, which have limited boundary types, and hence the context and term will only be able to communicate values of base type or function values.

Definition 5.1.2. A HOSC term $\Gamma \vdash M : \tau$ is *cr-free* if it does not contain occurrences of $\text{cont}_\sigma K$ and locations, and its boundary types are cont- and ref-free.

In the next section we study the trace language of FOSC under HOSC contexts. This trace language is used to characterise *contextual equivalence*, which is when two terms behave the same under the same contexts. First, we let $(M, h) \Downarrow$ denote that there exists a value V and heap h' such that $(M, h) \rightarrow^* (V, h')$. Then, we define contextual equivalence as below.

Definition 5.1.3. Given two HOSC terms $\Gamma \vdash M_1, M_2 : \sigma$, we say that M_1 *contextually approximates* M_2 , denoted $\Gamma \vdash M_1 \lesssim M_2$ if for all HOSC contexts $C[\sigma]$ such that $\vdash C[M_1], C[M_2] : \sigma'$, if $(C[M_1], \emptyset) \Downarrow$ then $(C[M_2], \emptyset) \Downarrow$. Furthermore, we say that the terms are *contextually equivalent*, denoted $\Gamma \vdash M_1 \simeq M_2$ whenever $\Gamma \vdash M_1 \lesssim M_2$ and $\Gamma \vdash M_2 \lesssim M_1$.

5.2 Trace semantics

In this section we introduce trace semantics for the language given above, as was established in [21]. This makes use of *operational game semantics*, in which the computations a program and its context engage in are modelled as interactions between a player (P) representing the term and an opponent (O) representing the context. Throughout this section, we will use the notational convention that the composition $\cdot$ implies the function domains are disjoint, and the union $\uplus$ implies the sets are disjoint.

Firstly, we note that the trace models are defined on an extension of the definition of FOSC. To the syntax we add function and continuation names $\text{FNames} = \uplus_{\sigma, \sigma'} \text{FNames}_{\sigma \rightarrow \sigma'}$, $\text{CNames} = \uplus_{\sigma} \text{CNames}_{\sigma}$ which are implicitly typed. We use c, d to refer to continuation names and f, g to refer to function names and write $c : \sigma$ and $f : \sigma \rightarrow \sigma'$ as shorthand for $c \in \text{CNames}_{\sigma}$, $f \in \text{FNames}_{\sigma \rightarrow \sigma'}$ respectively. Moreover, we let $\text{Names} = \text{CNames} \uplus \text{FNames}$, and let $\nu(X)$ represent the set of names used in X .

Furthermore, continuations are now represented in the syntax as staged continuations, $\text{cont}_{\sigma}(K, c)$, and the reduction rules for continuations are replaced with the rules:

$$\begin{aligned} (K[\text{call/cc}_{\tau}(x.M)], c, h) &\rightarrow (K[M\{\text{cont}_{\tau}(K, c)/x\}], c, h) \\ (K[\text{throw}_{\tau} V \text{ to } \text{cont}_{\tau}(K', c')], c, h) &\rightarrow (K'[V], c', h). \end{aligned}$$

For the rest of the reductions, if $(M, h) \rightarrow (N, h')$, then $(M, c, h) \rightarrow (N, c, h')$.

Now, arguments passed between the term and its context may refer to other names. To represent these, we make use of abstract values

$$A, B \triangleq f \mid () \mid \hat{n} \mid \mathbf{tt} \mid \mathbf{ff} \mid \langle A, B \rangle.$$

Then, given any concrete value V we can decompose it into its abstract value and a substitution using the following rules:

$$\begin{aligned} \mathbf{AVal}_{\sigma}(V) &\triangleq \{(V, \emptyset)\} \text{ for } \sigma \in \{\text{Unit}, \text{Bool}, \text{Int}\} \\ \mathbf{AVal}_{\sigma \rightarrow \sigma'}(V) &\triangleq \{(f, f \mapsto V) \mid f \in \text{FNames}_{\sigma \rightarrow \sigma'}\} \\ \mathbf{AVal}_{\sigma_1 \times \sigma_2}(\langle U_1, U_2 \rangle) &\triangleq \{(\langle A_1, A_2 \rangle, \gamma_1 \cdot \gamma_2) \mid (A_i, \gamma_i) \in \mathbf{AVal}_{\sigma_i}(U_i) \text{ for } i \in \{1, 2\}\} \end{aligned}$$

This is defined such that for any $(A, \gamma) \in \mathbf{AVal}_\sigma(V)$, $A\{\gamma\} = V$, that is substituting the names in A using γ gives us back our original term V .

We can now start to describe the traces which will represent an execution of our term in some context. Firstly, each action is either produced by the player (P) or the opponent (O). We start with an initial set of names, N_O , and furthermore new names are introduced in actions. We call the names introduced in a player action P-names, and the names introduced in an opponent action, as well as the initial set N_O of names as O-names.

Now, an action can either be a *question* or an *answer*. Given these can either be done by the opponent or the player, we end up with four actions, (PQ) , (PA) , (OQ) and (OA) . To ask a question, the player must query a function f introduced by the other player, and provide a value (represented using an abstract value A) and a context, which is used by the other player to return the answer. Conversely, to answer a question the player must return a value, again represented by an abstract value A to a given continuation. Here, new names are introduced in the moves (PQ) , (OQ) to represent the continuation, and may also be introduced in any action in the abstract value A provided. Note that we typically use $\mathbf{a}$ to denote actions. We list these actions and their representation below:

- *Player Answer (PA)*: $\bar{c}(A)$, where $\bar{c} : \sigma$ is an O-name, $A : \sigma$ is an abstract value possibly introducing new P-names.
- *Player Question (PQ)*: $\bar{f}(A, c)$, where $\bar{f} : \sigma \rightarrow \sigma'$ is an O-name, $c : \sigma'$ is a *new* P-name, and $A : \sigma$ is an abstract value, possibly introducing new P-names.
- *Opponent Answer (OA)*: $c(A)$, where $c : \sigma$ is a P-name, $A : \sigma$ is an abstract value possibly introducing new O-names.
- *Opponent Question (OQ)*: $f(A, c)$, where $f : \sigma \rightarrow \sigma'$ is a P-name, $c : \sigma'$ is a *new* O-name, and $A : \sigma$ is an abstract value, possibly introducing new O-names.

We now present the LTS used to generate these traces. There are two types of configurations: *active configurations*, which denote the term (player) is next to

$(P\tau)$	$\langle M, c, \gamma, \xi, \phi, h \rangle \xrightarrow{\tau} \langle N, c', \gamma, \xi, \phi, h' \rangle$ when $(M, c, h) \rightarrow (N, c', h')$
(PA)	$\langle V, c, \gamma, \xi, \phi, h \rangle \xrightarrow{\bar{c}(A)} \langle \gamma \cdot \gamma', \xi, \phi \uplus \nu(A), h \rangle$ when $c : \sigma, (A, \gamma') \in \mathbf{AVal}_\sigma(V)$
(PQ)	$\langle K[fV], c, \gamma, \xi, \phi, h \rangle \xrightarrow{\bar{f}(A, c')} \langle \gamma \cdot \gamma' \cdot [c' \mapsto K], \xi \cdot [c' \mapsto c], \phi', h \rangle$ when $f : \sigma \rightarrow \sigma', (A, \gamma') \in \mathbf{AVal}_\sigma(V), c' : \sigma', \phi' = \phi \uplus \nu(A) \uplus \{c'\}$
(OA)	$\langle \gamma, \xi, \phi, h \rangle \xrightarrow{c(A)} \langle K[A], c', \gamma, \xi, \phi \uplus \nu(A), h \rangle$ when $c : \sigma, A : \sigma, \gamma(c) = K, \xi(c) = c'$
(OQ)	$\langle \gamma, \xi, \phi, h \rangle \xrightarrow{f(A, c)} \langle VA, c, \gamma, \xi, \phi \uplus \nu(A) \uplus \{c\}, h \rangle$ when $f : \sigma \rightarrow \sigma', A : \sigma, c : \sigma', \gamma(f) = V$

Figure 5.3: $\mathcal{L}_{\text{HOSC}}$ LTS

move and *passive configurations*, which denote the context (opponent) is next to move. In both these configurations, we make use of the terms γ, ξ and ϕ . These are defined as follows:

- $\gamma : (\text{FNames} \rightarrow \text{Vals}) \cup (\text{CNames} \rightarrow \text{ECtxs})$ is a partial function that maps function names to values and continuation names to evaluation contexts. This will map the functions and continuations introduced by the term to their actual meanings.
- $\xi : \text{CNames} \rightarrow \text{CNames}$ is a partial function that maps continuation names to other continuation names. This will indicate for a given continuation to which continuation to pass the obtained value.
- $\phi \subseteq \text{Names}$ is the set of all names which have been introduced, both by the term and the context.

Then, the active configurations take the shape $\langle M, c, \gamma, \xi, \phi, h \rangle$, where M represents the current term being reduced and c, h represent the current continuation and heap. The passive configurations take the shape $\langle \gamma, \xi, \phi, h \rangle$.

Now, given a term we must first give values to the uninitialised variables in the term. We do this through a Γ -assignment. If $\Gamma \vdash M : \sigma$ is a HOSC cr-free term where $\Gamma = \{x_1 : \sigma_1, x_2 : \sigma_2, \dots, x_n : \sigma_n\}$, we define a Γ -assignment ρ of M

to be a mapping $\rho : \{x_1, \dots, x_n\} \rightarrow \text{AVals}$ such that $\rho(x_i) : \sigma_i$ and the names of the abstract values are disjoint. These names will make up the initial set of names N_O . Furthermore, we start with an empty γ, ξ as we do not have any P-names yet. Concretely, we define our initial configuration $C_M^{\rho, c}$ given the Γ -assignment ρ and an arbitrary $c : \sigma$ to be $\langle M\{\rho\}, c, \emptyset, \emptyset, N_O, \emptyset \rangle$.

From this point onwards, the term M will be reduced silently using the rules in Figure 5.2. This corresponds to $(P\tau)$. Eventually when reducing the term, we might encounter a term we can't reduce, as it depends on the behaviour of a function f introduced by the context. This can be an initial O-name or an O-name introduced later on by the context. In this case, the player plays (PQ) , providing the value of the function as an abstract value, initialising new names to represent it if necessary. As we do this, we update γ, ξ and ϕ accordingly.

From this point onwards, we are in a passive configuration so the context has control. The context can do one of two things: query the value of a function introduced by the term (OQ) or answer a question of the term (OA) by providing a value to a context. After this exchange, the control is back to the player which reduces the term it is given, either evaluating the answer to the opponent's question or continuing evaluation after having one of its questions answered. Again using $(P\tau)$, the term might reduce to a question (PQ) , or it might reduce to a value (PA) , in which case it returns the value to the appropriate context. This is the general idea behind the trace semantics, but the full reduction rules are specified in Figure 5.3.

Definition 5.2.1. Given configurations C, C' and trace $t = \mathbf{a}_1 \mathbf{a}_2 \dots \mathbf{a}_n$, we define $C \xrightarrow{t} C'$ if there exist $C_1, C'_1, \dots, C_n, C'_n$ such that $C \xrightarrow{\tau^*} C_1 \xrightarrow{\mathbf{a}_1} C'_1 \xrightarrow{\tau^*} C_2 \xrightarrow{\mathbf{a}_2} \dots \xrightarrow{\tau^*} C_n \xrightarrow{\mathbf{a}_n} C'_n \xrightarrow{\tau^*} C'$

Given these definitions, we can now define the trace of a configuration, hence the trace language of a term as below:

$$\mathbf{Tr}_{\text{HOSC}}(C) \triangleq \{t \mid \exists C' C \xrightarrow{t} C'\}.$$

Hence given $\Gamma \vdash M : \sigma$,

$$\mathbf{Tr}_{\text{HOSC}}(\Gamma \vdash M : \sigma) \triangleq \{((\rho, c), t) \mid \rho \text{ is a } \Gamma\text{-assignment, } c : \sigma, t \in \mathbf{Tr}_{\text{HOSC}}(C_M^{\rho, c})\}.$$

This trace model is useful as it can be used to determine contextual equivalence between our terms.

Theorem 5.2.2. *For any cr-free HOSC terms, $\Gamma \vdash M_1, M_2 : \sigma$, we have $\Gamma \vdash M_1 \lesssim M_2$ if and only if $\mathbf{Tr}_{\text{HOSC}}(\Gamma \vdash M_1 : \sigma) \subseteq \mathbf{Tr}_{\text{HOSC}}(\Gamma \vdash M_2 : \sigma)$.*

We will demonstrate the power of our nested data automata by translating the trace language of FOSC terms under HOSC contexts. To do this, we need to represent this trace in a way that is recognisable to the automata. We start by establishing bounds on the term size, so that it can be represented in the state space.

Lemma 5.2.3. *For any cr-free FOSC term M , there exists a bound $\text{TermBound}(M)$ such that if*

$$N \in \text{Reach}(M) = \{N : \mathbb{C}_M^{\rho, c} \xrightarrow{t} \langle N, c', \gamma, \xi, \phi, h \rangle\},$$

then $|N| \leq \text{TermBound}(M)$.

Proof. We do this by induction on the number of occurrences of while in the term M . When there are no occurrences, the result is true by [9], Corollary 4.7.

Now, suppose we have a term $R = C[\text{while } M \text{ do } N]$ with $n + 1$ occurrences of while. We define a term R' with n occurrences of while by replacing an instance of while M do N with $f(\lambda z. \text{if } M \text{ } N \text{ } ())$, that is we let $R' = C[f(\lambda z. \text{if } M \text{ } N \text{ } ())]$. We show that for every $T \in \text{Reach}(R)$ there exists $T' \in \text{Reach}(R')$ such that $|T| \leq \text{TermBound}(R') + |T'|$.

First note that if T is obtained from R by reducing C without touching the while loop, then there is an identical term T' obtained by reducing C in the same way. It is clear that in this case $|T| \leq |T'|$, and in this case the terms γ, ξ and ϕ are also in sync.

Eventually we could reach the two terms $T = K[\text{while } M \text{ do } N]$ and $T' = K'[f(\lambda z. \text{if } M \text{ } N \text{ } ())]$. At this point, the trace of R' will start to look different, as the player will announce the name f . We will denote the argument of f as $t = \lambda z. \text{if } M \text{ } N \text{ } ()$, and assign the name g to this function arbitrarily. Then, our transition in R' is

$$\langle K'[f(\lambda z. \text{if } M \text{ } N \text{ } ())], c, \gamma, \xi, \phi \rangle \xrightarrow{\bar{f}(g, c')} \langle \gamma \cdot [g \mapsto t] \cdot [c' \mapsto K'], \xi \cdot [c' \mapsto c], \phi \uplus \{c', g\} \rangle.$$

Note now that from R' we can call g as many times as we like using (OQ) with g as an argument. We always do this at least once, corresponding to checking the if condition for the first time. This is represented in the transition below,

$$\langle \gamma', \xi', \phi' \rangle \xrightarrow{g((), c'')} \langle t(), c'', \gamma', \xi', \phi' \rangle$$

where $\gamma' = \gamma \cdot [g \mapsto t] \cdot [c' \mapsto K']$, $\xi' = \xi \cdot [c' \mapsto c]$, $\phi' = \phi \uplus \{c', g\}$.

From this point onwards, we will be evaluating $t() = (\lambda z. \text{if } M \ N \ ())() \rightarrow \text{if } M \ N \ ()$, so we are no longer reducing the exact same term as for R . However, the reduction of these terms follow the exact same steps so long as we are still reducing the first expansion of the while loop. The difference between the two terms is of the surrounding context K and the appended while $M \text{ do } N$ in R . However, because of the term bound on R' and the correspondence between the contexts, this difference cannot exceed $\text{TermBound}(R')$, hence here our resulting term is still bounded by the size of the term in R' plus $\text{TermBound}(R')$.

We continue this simulation, noting that the behaviour in R' mirrors that of R when we are evaluating M and N . To keep this up, whenever M evaluates to true, we call g again using (OQ) after we finish evaluating the N of the previous iteration. By doing this, we ensure the two terms are always in sync and hence within the required bound.

The two terms will reduce differently when evaluating the continuations, as the c'' for R' refers to where we want the answer to the opponent question, meanwhile in R it is the original c . However in practice this does not pose any problems, as if the continuation was initialised before we entered the while loop, the terms go back into sync together as they were before the loop. Otherwise, the call/cc constructor must have been evaluated inside the while execution, and hence if we evaluate $\text{throw}_\tau V$ to $\text{cont}_\tau(K', c')$, we will end up again in the same situation inside the while execution.

Moreover, if M reduces to false the terms go back into sync as they were before the while loop, where R' will have some extra term g in γ that it can ignore. Now, this result implies that $2\text{TermBound}(R')$ suffices as a bound on the terms of R , hence we have proved our result by induction. $\square$

We also want a similar result for the heap akin to [9], Lemma 4.8, which states that for a term M , there exists a bound $\text{HeapBound}(M)$ such that the number of locations generated in consecutive τ -transitions is bounded by $\text{HeapBound}(M)$. However, this exact statement is not true for our program with loops. Take for example the term

$$M_1 \triangleq \text{while } \mathbf{tt} \text{ do let } x = \text{ref } 0 \text{ in } ().$$

It is clear that M_1 generates an unbounded number of references, and reduces using τ -transitions indefinitely. However, we can see that for this particular term, there is no way to access the created reference after the iteration ends.

It is important to note that references created during an iteration can sometimes still be accessed. Consider alternatively the term

$$M_2 \triangleq \text{while } \mathbf{tt} \text{ do let } x = \text{ref } 0 \text{ in } f(\lambda z. !x),$$

where f is an O-name. Since $\lambda z. !x$ is communicated to the context, the context can at any time access the value of x , even when the iteration is finished. However, we note that in this case our term no longer only reduces using τ -transitions, as when we encounter f , we transition using (PQ) .

Lemma 5.2.4. *If a location ℓ is created during an iteration of a while loop which terminates without calling (PQ) or (PA) , then ℓ is never accessed after the iteration terminates.*

Proof. Suppose that a location ℓ is initialised either in subterm M of the expansion of $M (N; \text{while } M \text{ do } N) ()$, or term N of $N; \text{while } M \text{ do } N$. It is clear from the reduction semantics that all instances of ℓ are contained in M or N respectively. Moreover, we do not have references of references so the value ℓ cannot be stored. Hence after the term finishes the iteration there are no more instances of ℓ that can be accessed. Since we are assuming $(PQ), (PA)$ were not called during the iteration itself, then we could not communicate ℓ to the context, hence after we finish the iteration ℓ is no longer accessible. $\square$

This means that resetting the heap to how it was originally when exiting a loop if and only if no P-move was made keeps the trace semantics the same, as we do not lose any information. Finally, it is immediate to see that if we do this

resetting, then between consecutive τ -transitions our heap only increases by some bounded amount. This is because our term size is bounded, and hence has a bounded number of terms in the form $\text{ref } V$. Moreover, at any point in time, for each loop in the term, our heap will store at most the new locations created in one iteration of the loop.

5.3 Replacing O-names with indices

We wish to use the same trick as in [9], and represent O-names as indices. To do this, we must first define the notion of a P-view. These determine the part of the trace which contains O-names the term can currently access.

Definition 5.3.1 (P-view).

$$\begin{aligned}
\text{Vis}_P(\epsilon) &\triangleq \epsilon \\
\text{Vis}_P(t \bar{f}(A, c) \ t' c(A')) &\triangleq \text{Vis}_P(t) \bar{f}(A, c) \ c(A') \\
\text{Vis}_P(t \bar{f}(A, c) \ t' g(A', c')) &\triangleq \text{Vis}_P(t) \bar{f}(A, c) \ t' g(A', c') & g \in \nu(A) \\
\text{Vis}_P(t \bar{c}(A) \ t' g(A', c')) &\triangleq \text{Vis}_P(t) \bar{c}(A) \ g(A', c') & g \in \nu(A) \\
\text{Vis}_P(t \ \mathbf{a}) &\triangleq \text{Vis}_P(t) \mathbf{a} & \mathbf{a} \text{ is a P-action}
\end{aligned}$$

As P-views determine the visible O-names, we can use indices instead of O-names and reuse these indices in different P-views. In [9], as stated in Corollary 4.5, the length of the P-views is conveniently bounded. This bound implies that the indices we need are bounded by some constant. However, adding while loops, the P-views are no longer of some bounded length.

Example 5.3.2. Consider the term

$$\{f : \text{Int} \rightarrow \text{Int} \rightarrow \text{Unit}\} \vdash \text{while } \mathbf{tt} \text{ do } (\text{let } g = f\hat{0} \text{ in } g\hat{1}).$$

Clearly, in this term we repeatedly execute the code $\text{let } g = f\hat{0} \text{ in } g\hat{1}$. The interaction for the body of the loop starts with the term asking the context what the result of evaluating f on $\hat{0}$ is. Since $f : \text{Int} \rightarrow \text{Int} \rightarrow \text{Unit}$, the context must reply with another function, of type $\text{Int} \rightarrow \text{Unit}$, hence introducing some

new name g' . Then, the term further asks the context what the result of evaluating g' at value $\hat{1}$ is. In the loop, every time this process is repeated, a new name g is generated as a response to the player's question. This generates traces of the form $\bar{f}(\hat{0}, c_1) \ c_1(g_1) \ \bar{g}_1(\hat{1}, c'_1) \ c'_1(()) \ \bar{f}(\hat{0}, c_2) \ c_2(g_2) \ \bar{g}_2(\hat{1}, c'_2) \ c'_2(()) \dots \bar{f}(\hat{0}, c_n) \ c_n(g_n) \ \bar{g}_n(\hat{1}, c'_n) \ c'_n(())$ for $n \in \mathbb{N}$.

In fact, the P-view for this trace consists of the whole view, hence the size of the P-view for this term grows unboundedly with the number of iterations considered. We can see this, as representing our trace with indices, representing f as 1, c as 2 we get the trace $\bar{1}(\hat{0}, c_1) \ c_1(3) \ \bar{3}(\hat{1}, c'_1) \ c'_1(()) \ \bar{1}(\hat{0}, c_2) \ c_2(4) \ \bar{4}(\hat{1}, c'_2) \ c'_2(()) \dots \bar{1}(\hat{0}, c_n) \ c_n(n+2) \ \bar{n+2}(\hat{1}, c'_n) \ c'_n(())$.

However, if we look closely at the actual execution of the term, once we repeat the loop, we no longer need access to the name of the previous iteration. Hence it would be reasonable to reuse the same variable name for $g_1, g_2, \dots, g_n$, and this is in fact how we deal with the unbounded P-view problem.

To translate our terms to using indices, we first define some auxiliary notation to translate between names and indices. It is important to notice that our indices are no longer implicitly typed as our names were. To rectify this, we embed the types into the syntax, translating any instance of the name f for $f : \sigma$ into $i : \sigma$. Moreover, for $(A^I, k) \in \mathbf{IAVal}_\sigma(k)$, we let $U(A^I)$ be the untyped version of A^I .

Definition 5.3.3 (Indexed Abstract Value).

$$\begin{aligned} \mathbf{IAVal}_\sigma(k) &\triangleq \{(V, k) \mid V : \sigma\} \text{ for } \sigma \in \{\text{Unit}, \text{Bool}, \text{Int}\} \\ \mathbf{IAVal}_{\sigma \rightarrow \sigma'}(k) &\triangleq \{(k : \sigma \rightarrow \sigma', k+1)\} \\ \mathbf{IAVal}_{\sigma \times \sigma'}(k) &\triangleq \{(\langle A_1, A_2 \rangle, k_2) \mid (A_1, k_1) \in \mathbf{IAVal}_\sigma(k) \text{ and } (A_2, k_2) \in \mathbf{IAVal}_{\sigma'}(k_1)\} \end{aligned}$$

To handle the loop resetting action, we need special behaviour for our while loops and a way to store the domain of the current heap and the next index. For this reason, we use a stack of integer pairs ζ . Additionally, for reasons explained previously, we will not reset the heap when we have communicated with the context inside the loop. To signify that our heap should not be reset, we use the indicator $\#$. For this purpose, we define

$$\text{saveHeap}(\zeta) = \begin{cases} (\#, k). \zeta' & \text{if } \zeta = (m, k). \zeta'; \\ \zeta & \text{otherwise.} \end{cases}$$

$$\begin{array}{l|l}
(P\tau) & \langle M, (i : \sigma), k, \zeta, \gamma, \xi, \phi, \eta, h \rangle \xrightarrow{\tau} \langle N, (i' : \sigma'), k', \zeta', \gamma, \xi, \phi, \eta, h \rangle \\
& \text{when } (M, (i : \sigma), h, k, \zeta) \rightarrow (N, (i' : \sigma'), h', k', \zeta') \\
(PA) & \langle V, (i : \sigma), k, \zeta, \gamma, \xi, \phi, \eta, h \rangle \xrightarrow{\bar{i}(A)} \langle \gamma \cdot \gamma', \xi, \phi \uplus \nu(A), \eta', h \rangle \\
& \text{when } (A, \gamma') \in \mathbf{AVal}_\sigma(V), \eta' = \eta \cdot [\nu(A) \mapsto (k, \text{saveHeap}(\zeta))] \\
(PQ) & \langle K[(i : \tau)V], (j : \tau'), k, \zeta, \gamma, \xi, \phi, \eta, h \rangle \xrightarrow{\bar{i}(A, c')} \langle \gamma \cdot \gamma' \cdot [c' \mapsto K], \xi', \phi', \eta', h \rangle \\
& \text{when } \tau = \sigma \rightarrow \sigma', (A, \gamma') \in \mathbf{AVal}_\sigma(V), c' : \sigma', \xi' = \xi \cdot [c' \mapsto (j : \tau')], \\
& \phi' = \phi \uplus \nu(A) \uplus \{c'\}, \eta' = \eta \cdot [\nu(A) \mapsto (k, \text{saveHeap}(\zeta))] \\
(OA) & \langle \gamma, \xi, \phi, \eta, h \rangle \xrightarrow{c(U(A^I))} \langle K[A^I], (i : \tau), k', \zeta, \gamma, \xi, \phi', \eta, h \rangle \\
& \text{when } c : \sigma, (k, \zeta) = \eta(c), (A^I, k') \in \mathbf{IAVal}_\sigma(k), \gamma(c) = K, \xi(c) = (i : \tau), \\
& \phi' = \phi \uplus \nu(A^I) \\
(OQ) & \langle \gamma, \xi, \phi, \eta, h \rangle \xrightarrow{f(U(A^I), i)} \langle VA^I, (i : \sigma'), i + 1, \zeta, \gamma, \xi, \phi \uplus \nu(A^I) \uplus \{i\}, \eta, h \rangle \\
& \text{when } f : \sigma \rightarrow \sigma', (k, \zeta) = \eta(f), (A^I, i) \in \mathbf{IAVal}_\sigma(k), \gamma(f) = V
\end{array}$$

Figure 5.4: Modified LTS

which updates our heap ζ accordingly when a P-move is made. The number of nesting in any given term is bounded, as it is limited by the number of while loops present, which means our stack has a bounded depth. Moreover, the index is bounded because of the resetting. The size of the heap is not bounded yet, as our heap can grow unboundedly. We will handle this later on in our automata.

Now, we need to be able to recognise when we enter a loop and when we leave it. We conveniently already have an indication of where the loop starts because of the while keyword, however we need some way to delimit the end of the loop. We do this with the keyword exit, modifying the reduction to be while M do $N \rightarrow$ if M (N ; exit; while M do N) ($\cdot$). Then, entering the while loop, we save the current next index i , and the domain of the heap $\text{dom}(h)$ onto the stack ζ . When we exit the loop, that is, when we see the keyword exit, we pop the value of the stack, keeping the heap the same if and only if $\#$ is at the top of the stack. Note that by $h_{\leq m}$ we mean the mapping h restricted only to the domain of integers $\leq m$.

We also need to keep track of the fact that when we call a continuation, we might end up in a different while loop as to before. To make sure that exit actually corresponds to the index at the beginning of the while loop we are exiting, we also

store the loop stack in the staged continuation $\text{cont}_\sigma(K, c)$ as $\text{cont}_\sigma(K, c, \zeta)$. Hence we have the following operational semantics:

- $(K[\text{while } M \text{ do } N], (i : \sigma), h, k, \zeta) \rightarrow (\text{if } M \text{ } N; \text{exit}; \text{while } M \text{ do } N \text{ } (), (i : \sigma), h, k, (\text{dom}(h), k). \zeta),$
- $(K[\text{exit}], (i : \sigma), h, k, (m, k'). \zeta) \rightarrow (K[()], (i : \sigma), h_{\leq m}, k', \zeta) \text{ for } m \in \mathbb{N},$
- $(K[\text{exit}], (i : \sigma), h, k, (\#, k'). \zeta) \rightarrow (K[()], (i : \sigma), h, k', \zeta).$
- $(K[\text{call/cc}_\tau(x.M)], (i : \sigma), h, k, \zeta) \rightarrow (K[M\{\text{cont}_\tau(K, (i : \sigma), \zeta)/x\}], (i : \sigma), h, k, \zeta),$
- $(K[\text{throw}_\tau V \text{ to } \text{cont}_\tau(K', (i' : \sigma'), \zeta')], (i : \sigma), h, k, \zeta) \rightarrow (K'[V], (i' : \sigma'), h, k, \zeta').$

Otherwise, transitions remain as usual, keeping ζ the same and updating k to always indicate the next index free for O-names. This process is described in detail in [9].

Given this, we can now describe the game semantics of our indexed language, replacing O-names with indices and resetting loops. We keep track of the next available index k in our state space, and also keep track of a partial map $\eta : \text{Names} \rightarrow \mathbb{N} \times ((\mathbb{N} \cup \{\#\}) \times \mathbb{N})^*$ which maps introduced P-names to the index k and the stack ζ at which the name was created. This translation can be found in Figure 5.4. Note that we only use abstract values with indices when dealing with O-moves, as we do not reuse P-names. In fact it is not possible to reuse P-names, as the context is taken to be from HOSC and can memorise the P-names.

Now, the initial configuration $C_M^{\rho, c}$ given the Γ -assignment ρ and an arbitrary $c : \sigma$ is set to be $\langle \text{Ind}[M\{\rho\}], (i : \sigma), i+1, \epsilon, \emptyset, \emptyset, N_O, \emptyset \rangle$, where $\text{Ind}[M\{\rho\}]$ is obtained by replacing each uninitialised variable with $(j : \sigma)$ for a unique $j \in \mathbb{N}$, starting from 1 and ending at $i - 1$, and $N_O = \{1, \dots, i\}$. This corresponds to converting O-names to indices and starting with the next free index as k .

Definition 5.3.4. We define the indexed trace language from a configuration as expected, letting $\mathbf{Tr}_{\text{ind}}(C) \triangleq \{t \mid \exists C' \ C \xrightarrow{t} C'\}$. Moreover, given $\Gamma \vdash M : \sigma$, the *indexed trace language* of a cr-free FOSC term $\Gamma \vdash M : \sigma$ is

$$\mathbf{Tr}_{\text{ind}}(\Gamma \vdash M : \sigma) \triangleq \{((\rho, c), t) \mid \rho \text{ is a } \Gamma\text{-assignment, } c : \sigma, t \in \mathbf{Tr}_{\text{ind}}(C_M^{\rho, c})\}.$$

Now, by [9], Lemma 5.10, indices used in traces without while loops are bounded, hence since we reset the index after every loop iteration the following lemma holds.

Lemma 5.3.5. *For any cr -free FOSC term $\Gamma \vdash M : \sigma$, there exists a bound $\text{MaxInd}(M)$ such that all indices appearing in $\text{Tr}_{ind}(\Gamma \vdash M : \sigma)$ are bounded by $\text{MaxInd}(M)$.*

Remark 5.3.6. Since we reset the next index whenever we encounter a loop, we sometimes unintentionally expose where the while loops are. This means that two terms might have the same trace in the original game semantics, but different ones in our indexed semantics. For example, a loop that iterates n times and creates a new name in each loop is equivalent to creating a name n times, however the traces of these terms will be different as the first program will reuse the same name while the second program will not. We see this in a concrete example below.

Example 5.3.7. Consider the term

$$\{f : \text{Int} \rightarrow \text{Int} \rightarrow \text{Unit}\} \vdash \text{while } \mathbf{tt} \text{ do } (\text{let } g = f\hat{0} \text{ in } g\hat{1})$$

as defined earlier. Using our method of reusing names, this will have the trace $\bar{1}(\hat{0}, c_1) \ c_1(3) \ \bar{3}(\hat{1}, c'_1) \ c'_1(()) \ \bar{1}(\hat{0}, c_2) \ c_2(3) \ \bar{3}(\hat{1}, c'_2) \ c'_2(()) \dots \bar{1}(\hat{0}, c_n) \ c_n(3) \ \bar{3}(\hat{1}, c'_n) \ c'_n(())$. Now consider the term

$$\{f : \text{Int} \rightarrow \text{Int} \rightarrow \text{Unit}\} \vdash \text{while } \mathbf{tt} \text{ do } (\text{let } g_1 = f\hat{0} \text{ in } (g_1\hat{1}; \text{let } g_2 = f\hat{0} \text{ in } g_2\hat{1})).$$

Clearly, this term does the exact same thing, as it repeatedly asks for the function resulting from $f\hat{0}$, then calls this returned function with the parameter $\hat{1}$. However, the trace for this term is $\bar{1}(\hat{0}, c_1) \ c_1(3) \ \bar{3}(\hat{1}, c'_1) \ c'_1(()) \ \bar{1}(\hat{0}, c_2) \ c_2(4) \ \bar{4}(\hat{1}, c'_2) \ c'_2(()) \bar{1}(\hat{0}, c_3) \ c_3(3) \ \bar{3}(\hat{1}, c'_3) \ c'_3(()) \dots$, alternating between the label 3 and 4, hence the two generated indexed traces are not in fact equal.

From the fact that our indexed translation can be used to reconstruct the original trace semantics, we can deduce that trace inclusion for our indexed translation implies contextual approximation of terms. The converse is not true, as was shown above.

Lemma 5.3.8. *If $\text{Tr}_{ind}(\Gamma \vdash M_1 : \sigma) \subseteq \text{Tr}_{ind}(\Gamma \vdash M_2 : \sigma)$, then $\Gamma \vdash M_1 \lesssim M_2$.*

5.4 Translation to NDCMA

The main challenge in representing the trace language above as an automaton is representing our language as a data language, that is a language over $(\Sigma \times \mathcal{D})^*$ for a nested data set $\mathcal{D}$. Fortunately, this process has already been done in [9], and we can reuse the main ideas.

5.4.1 Representing plays

Firstly, we notice that each move from PA, PQ, OA, OQ may use any number of names. For this purpose, we decompose the move into multiple elements of $\Sigma \times \mathcal{D}$. To do this, we use patterns of the form

$$A_{\square} \triangleq \square \mid () \mid \hat{n} \mid \mathbf{tt} \mid \mathbf{ff}$$

where n is any integer between 0 and max. Then we can convert from a pattern and a list of names to a value and vice versa (using indices instead of names) as expected, using the following decomposition and reconstruction rules

$$\begin{array}{ll} \text{Decom}(V) \triangleq (V, \epsilon) & V : \{\text{Unit}, \text{Int}, \text{Bool}\} \\ \text{Decom}(V) \triangleq (\square, V) & V : \sigma \rightarrow \sigma' \\ \text{Decom}(\langle V_1, V_2 \rangle) \triangleq (\langle P_1, P_2 \rangle, \bar{U}_1 \bar{U}_2) & (P_i, \bar{U}_i) = \text{Decom}(V_i) \\ \\ \text{Recon}(V, \epsilon) \triangleq V & V : \{\text{Unit}, \text{Int}, \text{Bool}\} \\ \text{Recon}(\square, (i : \sigma)) \triangleq (i : \sigma) & \sigma = \sigma' \rightarrow \sigma'' \\ \text{Recon}(\langle V_1, V_2 \rangle, \bar{I}_1 \bar{I}_2) \triangleq \langle U_1, U_2 \rangle & U_i = \text{Recon}(P_i, \bar{I}_i), |I_i| = \#P_i. \end{array}$$

We denote the number of squares in $A_{\square}$ as $\#A_{\square}$ and note that for a reconstruction to be valid, the list provided must be exactly of length $\#A_{\square}$.

Since the indices we use to represent O-names are bounded, we can assume that our automata is already initialised with the indices in memory as level 0 data values with no memory. This tweak does not change any properties of our NDCMA, as it only means we initialise the class memory function. We could deal with this with regular NDCMA using some initial padding, as we did with Petri nets.

Now, to represent an action, we use tags Pcall, Ocall to denote (PQ) , (OQ) and Pret, Oret to denote (PA) , (OA) . We start an action by denoting its type, and the pattern $A_\square$ of the value being sent, in the form $\text{Action}(A_\square)$. Each answer is in response to a question, and hence is sent to a particular continuation, and each question necessarily calls some function, hence we use the data value of this letter to represent this name. All together, our header value has the shape $(\text{Action}(A_\square), d)$.

This is followed by a number $\#A_\square$ of parameters, tagged as Parg or Oarg, where the data value indicates the name of the parameter inputted. This is enough to represent our return (answer) moves. For call (question) moves, we need to further specify the name of the continuation being used, which we do using the tag Pcont and Ocont. For example, the (indexed) P-move $\bar{4}(\langle f, g \rangle, c)$ would be translated as

$$(\text{Pcall}(\langle \square, \square \rangle), \tilde{4}), (\text{Parg}, d_1), (\text{Parg}, d_2), (\text{Pcall}, d_3).$$

5.4.2 Handling the tree structure

As in [9], we want each branch of the memory to represent a P-view. This is done by maintaining ‘ghost values’ which in essence keep track of the desired tree structure. Essentially, each P-move is preceded by (Gopen, o_1) and succeeded by (Gclose, o_1) , where o_1 is a ‘ghost’ data value. These ghost values are related as to represent the P-views, enforcing that when a P-move is played, its ghost value is the child of the P-move last played in its view.

This was done because the trace language satisfies *P-visibility* [21], meaning that when the player makes use of a reference, this must lie in the P-view of the move played. In bounded NDCMA, we have access to all the ancestors of a new data value, hence access to the whole branch. This means that the heap created within the P-view of a move can be accessed by the move, which is enough by P-visibility.

P-visibility still holds for our language, and in fact we adapt the construction in [9], letting each branch represent an entire P-view. However, as we saw in Example 5.3.2, this P-view can be unbounded, so this is not enough for us to have a finite description. Fortunately, we also have the fact that our P-moves can only access the references within the scope of the move. In particular, we cannot access references created in different iterations of a loop. We reflect this by using the

memory of a ghost value to store only the moves in the same iteration. This gives us a finite memory as desired.

In practice, this means that after we reset a loop iteration we forget the moves from that loop. This is done by keeping track of the memory size d both of the current state and at the relevant points of a term, using a stack as we did for the index. Hence we now use the stack ζ to contain tuples of heap size, index and memory as indicated below.

- $(K[\text{while } M \text{ do } N], (i : \sigma), h, k, d, \zeta) \rightarrow (\text{if } M \text{ } N; \text{exit}; \text{while } M \text{ do } N \text{ exit}, (i : \sigma), h, k, (\max(h), k, d). \zeta),$
- $(K[\text{exit}], (i : \sigma), h, k, d, (m, k', d'). \zeta) \rightarrow (K[], (i : \sigma), h_{\leq m}, k', d', \zeta)$ if $m \in \mathbb{N}$,
- $(K[\text{exit}], (i : \sigma), h, k, d, (\#, k', d'). \zeta) \rightarrow (K[], (i : \sigma), h, k', d', \zeta)$,
- $(K[\text{call/cc}_\tau(x.M)], (i : \sigma), h, k, d, \zeta) \rightarrow (K[M\{\text{cont}_\tau(K, (i : \sigma), \zeta)/x\}], (i : \sigma), k, d, \zeta),$
- $(K[\text{throw}_\tau V \text{ to } \text{cont}_\tau(K', (i' : \sigma'), \zeta')], (i : \sigma), h, k, d, \zeta) \rightarrow (K'[V], (i' : \sigma'), h, k, \zeta').$

Given this, we can now store our heap inside the ‘ghost values’ representing our P-moves. We note again that the heap is taken to be defined on a contiguous range starting from 0. The heap will be represented in the class memory as disjoint maps $h_1, \dots, h_k$, where each branch stores the relevant heap distributively according to the P-view. Now, when we want to access the heap of a particular P-view ending with some ghost value o , we take the heap to be the union of the heap fragments stored in o and its memory $m(o)$, $h_1, h_2, \dots, h_k$, letting $h = h_1 \cdot h_2 \cdot \dots \cdot h_k$. This represents the entire visible heap for the upcoming P-move because of our reasoning above, hence suffices for our move.

When we finish reducing our term, our heap h is updated. Different references may have different values now, and moreover the heap can be enlarged. We represent this update using the function $\text{Update}(h, (\bar{h}))$, as in [9], where $m = \max\{\text{dom } h_i : h = h_1 \dots h_n\}$.

$$\begin{aligned} \text{Update}(h, (h_1)) &\triangleq (h) \\ \text{Update}(h, (h', \bar{h})) &\triangleq (h_{>m}, \text{Update}(h'_{\leq m}, \bar{h})) \end{aligned}$$

An important observation for our implementation is that when we update the heap, the newer values are always introduced in the newer heap value, which is tied to the new ghost value. In fact, whenever a reference is introduced in a P-move, its reference will always be stored in the corresponding ghost name. While update is defined on $(h_1, \dots, h_k)$, the usage will always have $h_1 = \emptyset$, and this heap will as per the definition store the new names. Moreover, the way update is defined, if a name is defined at some position it remains that way. Altogether, this means that different iterations of a while loop that introduce new names cannot interfere with each other, and that the heap obtained by taking the union is always faithful.

Given an indexed cr-free FOSC term $\Gamma \vdash M : \sigma$ and a Γ assignment ρ , we let $\mathcal{A}_M^{\rho, \sigma}$ be the weak deterministic r-NDCMA with initial state Init , initialised with data values $\tilde{1}, \tilde{2}, \dots, \tilde{n}$ where $n = \text{MaxInd}(M)$ and with the transitions as specified in 5.5. Our state space is finite for the same reasons as in [9], with the extra conditions that the depth and indices are bounded because of the fact that we reset them when we exit a loop iteration. Note that our heap h has a bounded size because of the bounded memory size, and the fact that each P-move only increases the heap size by some bounded number, hence the visible heap is also always bounded.

Our transitions are mostly taken from [9], with the addition of the stack and memory parameters. We note some differences here, in particular with the transitions (ϵ) and (Gopen) , and managing the heap. In (Oret) and (Ocall) , we obtain the heap from the visible parts of the P-view of our next move, and store it in the state, marking the ‘head’ of the P-view with $\dagger$. We then reduce the term with (ϵ) , updating the heap as we go along until we reach a value or an O-name being called. Then, we create the new ghost value with (Gopen) which represents the P-move. Here we update the heap fragments, noting that we could have updated the entire visible heap. We also set the memory of the ghost value to reflect the current accessible heap of the P-view from the P-move, according to d . Note that the order of (ϵ) and (Gopen) here is different from [9], as we need to have reduced the term to determine its memory.

Also note that when comparing directly with [9], our data annotations look reversed — this is simply the result of our notation of putting the new data value at the beginning of the tuple, rather than the end.

Lemma 5.4.1. $\mathcal{A}_M^{\rho,\sigma}$ faithfully represents $\mathbf{Tr}_{ind}(C_M^{\rho,\sigma})$ as translated to a data language.

5.5 Further work and limitations

In [9], the translation to automata was used to show decidability of the fragment of HOSC. However in our case we are unable to do this. This is because of two main obstacles, one relating to the translation and another to the NDCMA machinery.

Firstly, our translation to indexed O-names does preserve equality of traces, as we saw in Remark 5.3.6. One potential way we might be able to rectify this is to pick new indices non-deterministically, that is if n is an unused index, we can use n . This would require some extra bookkeeping as we need to keep track of used/unused names instead of indices, however this method would still be bounded, as we could take the maximum bound for indices of both terms. Note that this would not actually introduce non-determinism into our recognising automata, as to be deterministic we require our automata to act the same given the same label *and* data value, and our indices are in fact data values. The main fault in this approach is that one term might be able to reuse the same name, while another term is unable to. This would still cause a divergence in their indexed trace models. Perhaps it could be possible to circumvent this issue by avoiding repeating indices as much as possible, but this route is still unclear.

However, even if we did have a completely faithful representations, we still have limitations to the NDCMA. In particular, we do not yet know if equivalence of weak deterministic r-NDCMA is decidable. This may be unlikely given the undecidability results we have found so far, especially given inclusion is already known to be undecidable. An interesting direction for future work is to see if this translation can be done in the other way, in particular seeing if we can use a similar construction as to Theorem 4.3.9 to show that inclusion for the trace languages of HOSC[FOSC] is undecidable.

(Init)	Initial	$\xrightarrow[\emptyset^\dagger, m=0]{\text{Gopen}, \perp}$	$(M\{\rho\}, (i : \sigma), \emptyset, i + 1, 1, \epsilon, \text{Red})$
	with $i = \nu(\rho) + 1$		
(ϵ)	$(N, (i : \tau), h, k, d, \zeta, \text{Red})$ when $(N, (i : \tau), h, k, d, \zeta) \rightarrow (N', (i' : \tau'), h', k', d', \zeta')$	$\xrightarrow{\epsilon}$	$(N', (i' : \tau'), h', k', d', \zeta', \text{Red})$
(Gopen)	$(N, (i : \tau), h, k, d, \gamma, \text{Red})$ when $N = V$ or $N = K[(i : \tau)V]$ and $(h', h'_1, \dots, h'_n) = \text{Update}(h, (\emptyset, h_1, \dots, h_n))$	$\xrightarrow[(h'^\dagger, h'_1, \dots, h'_n), m=d]{\text{Gopen}, (\perp, h_1^\dagger, \dots, h_n)}$	$(N, (i : \tau), k, d + 1, \zeta)$
(Pcall)	$(K[(i : \sigma)V], (j : \sigma'), k, d, \zeta)$ when $(A_\square, \bar{V}) = \text{Decom}(V)$	$\xrightarrow[i]{\text{Pcall}(A_\square), i}$	$(K, (j : \sigma'), k, d, \zeta, \bar{V}, \text{Parg})$
(PargC)	$(K, (i : \sigma), k, d, \zeta, V.\bar{V}, \text{Parg})$ with $S = (V, k, d, \zeta)$	$\xrightarrow[(S, h_1^\dagger, \dots, h_n), m=d+1]{\text{Parg}, (\perp, h_1^\dagger, \dots, h_n)}$	$(K, (i : \sigma), k, d, \zeta, \bar{V}, \text{Parg})$
(PargC ϵ)	$(K, (i : \sigma), k, d, \zeta, \epsilon, \text{Parg})$	$\xrightarrow{\epsilon}$	$(K, (i : \sigma), k, d, \zeta, \text{Pcont})$
(Pcont)	$(K, (i : \sigma), k, d, \zeta, \text{Pcont})$ with $S = (K, (i : \sigma), k, d, \zeta)$	$\xrightarrow[(S, h_1^\dagger, \dots, h_n), m=d+1]{\text{Pcont}, (\perp, h_1^\dagger, \dots, h_n)}$	Gclose
(Pret)	$(V, (i : \sigma), k, d, \zeta)$ when $(A_\square, \bar{V}) = \text{Decom}(V)$	$\xrightarrow[i]{\text{Pret}(A_\square), i}$	$(\bar{V}, k, d, \zeta)$
(PargR)	$(V\bar{V}, k, d, \zeta)$ with $S = (V, k, d, \zeta)$	$\xrightarrow[(S, h_1^\dagger, \dots, h_n), m=d+1]{\text{Parg}, (\perp, h_1^\dagger, \dots, h_n)}$	$(\bar{V}, k, d, \zeta)$
(PargR ϵ)	(ϵ, k, d, ζ)	$\xrightarrow{\epsilon}$	Gclose
(Gclose)	Gclose	$\xrightarrow[(h_1, \dots, h_n)]{\text{Gclose}, (h_1^\dagger, \dots, h_n)}$	Ostate
(Ocall)	Ostate when $S = (V, k, d, \zeta), V : \sigma' \rightarrow \sigma'', \bar{\sigma} \vdash A_\square : \sigma', \bar{i} = (k, \dots, k' - 1), k' = k + \#A_\square$ $U = \text{Recon}(A_\square, (\bar{i} : \sigma_i)), h = h_1 \cdot h_2 \cdot \dots \cdot h_n$	$\xrightarrow[(S, h_1^\dagger, \dots, h_n)]{\text{Ocall}(A_\square), (S, h_1, \dots, h_n)}$	$(VU, (k' : \sigma''), h, \bar{i}, k', k' + 1, d, \zeta, \text{Oarg})$
(OargC)	$(N, (k : \sigma), h, \bar{i}, k, k + 1, d, \zeta, \text{Oarg})$	$\xrightarrow[i]{\text{Oarg}, i}$	$(N, (k : \sigma), h, \bar{i}, k, k + 1, d, \zeta, \text{Oarg})$
(OargC ϵ)	$(N, (k : \sigma), h, \epsilon, k, k + 1, d, \zeta, \text{Oarg})$	$\xrightarrow{\epsilon}$	$(N, (k : \sigma), h, k, k + 1, d, \zeta, \text{Ocont})$
(Ocont)	$(N, (k : \sigma), h, k, k + 1, d, \zeta, \text{Ocont})$	$\xrightarrow[k]{\text{Ocont}, k}$	$(N, (k : \sigma), h, k, k + 1, d, \zeta, \text{Red})$
(Oret)	Ostate when $S = (K, (i : \sigma), k, d, \zeta), K[\sigma'], \bar{\sigma} \vdash A_\square : \sigma', \bar{i} = k \dots k' - 1, k' = k + \#A_\square$ $U = \text{Recon}(A_\square, (\bar{i} : \sigma_i)), h = h_1 \cdot h_2 \cdot \dots \cdot h_n$	$\xrightarrow[(S, h_1^\dagger, \dots, h_n)]{\text{Oret}(A_\square), (S, h_1, \dots, h_n)}$	$(K[U], (i : \sigma), h, \bar{i}, k', d, \zeta, \text{Oarg})$
(OargR)	$(N, (k : \sigma), h, \bar{i}, d, \zeta, k, \text{Oarg})$	$\xrightarrow[i]{\text{Oarg}, i}$	$(N, (k : \sigma), h, \bar{i}, d, \zeta, k, \text{Oarg})$
(OargR ϵ)	$(N, (k : \sigma), h, \epsilon, d, \zeta, k, \text{Oarg})$	$\xrightarrow{\epsilon}$	$(N, (k : \sigma), h, d, \zeta, k, \text{Red})$

Figure 5.5: Transitions of the r -NDCMA $\mathcal{A}_M^{\rho, \sigma}$.

Bibliography

- [1] Toshiro Araki and Tadao Kasami. Some decision problems related to the reachability problem for Petri nets. *Theoretical Computer Science*, 3(1):85–104, 1976.
- [2] Henrik Björklund and Mikołaj Bojańczyk. Shuffle expressions and words with nested data. In Luděk Kučera and Antonín Kučera, editors, *Mathematical Foundations of Computer Science 2007*, pages 750–761, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
- [3] Henrik Björklund and Thomas Schwentick. On notions of regularity for data languages. *Theoretical Computer Science*, 411(4):702–715, 2010. Fundamentals of Computation Theory.
- [4] Mikołaj Bojańczyk, Laurent Braud, Bartek Klin, and Sławomir Lasota. Towards nominal computation. *SIGPLAN Not.*, 47(1):401–412, January 2012.
- [5] Mikołaj Bojańczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data words. *ACM Trans. Comput. Logic*, 12(4), July 2011.
- [6] Mikołaj Bojańczyk, Bartek Klin, and Sławomir Lasota. Automata with group actions. In *2011 IEEE 26th Annual Symposium on Logic in Computer Science*, pages 355–364, 2011.
- [7] Mikołaj Bojańczyk, Bartek Klin, and Sławomir Lasota. Automata theory in nominal sets. *Logical Methods in Computer Science*, Volume 10, Issue 3, August 2014.
- [8] Mikołaj Bojańczyk, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data trees and XML reasoning. *J. ACM*, 56(3), May 2009.
- [9] Benedict Bunting and Andrzej Murawski. Contextual equivalence for state and control via nested data. In *Proceedings of the 39th Annual ACM/IEEE*

Symposium on Logic in Computer Science, LICS '24, New York, NY, USA, 2024. Association for Computing Machinery.

- [10] Taolue Chen, Fu Song, and Zhilin Wu. *Formal Reasoning on Infinite Data Values: An Ongoing Quest*, pages 195–257. Springer International Publishing, Cham, 2017.
- [11] Vincenzo Ciancia and Matteo Sammartino. A class of automata for the verification of infinite, resource-allocating behaviours. In Matteo Maffei and Emilio Tuosto, editors, *Trustworthy Global Computing*, pages 97–111, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
- [12] Conrad Cotton-Barratt. *Using Class Memory Automata in Algorithmic Game Semantics*. Doctoral dissertation, University of Oxford, 2016.
- [13] Conrad Cotton-Barratt, David Hopkins, Andrzej Murawski, and C.-H. Luke Ong. Fragments of ML decidable by nested data class memory automata. In *International Conference on Foundations of Software Science and Computation Structures*, pages 249–263. Springer, 2015.
- [14] Conrad Cotton-Barratt, Andrzej Murawski, and C.-H. Luke Ong. Weak and nested class memory automata. In Adrian-Horia Dediu, Enrico Formenti, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications*, pages 188–199, Cham, 2015. Springer International Publishing.
- [15] Normann Decker, Peter Habermehl, Martin Leucker, and Daniel Thoma. Ordered navigation on multi-attributed data words. In Paolo Baldan and Daniele Gorla, editors, *CONCUR 2014 – Concurrency Theory*, pages 497–511, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
- [16] Diego Figueira. Alternating register automata on finite words and trees. *Log. Methods Comput. Sci.*, 8(1), 2012.
- [17] Alain Finkel and Philippe Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1):63–92, 2001. ISS.
- [18] Radu Grigore, Dino Distefano, Rasmus Lerchedahl Petersen, and Nikos Tzevelekos. Runtime verification based on register automata. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 260–276, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

- [19] Christoph Haase, Sylvain Schmitz, and Philippe Schnoebelen. The power of priority channel systems. In Pedro R. D’Argenio and Hernán Melgratti, editors, *CONCUR 2013 – Concurrency Theory*, pages 319–333, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- [20] Piotr Hofman, Richard Mayr, and Patrick Totzke. Decidability of weak simulation on one-counter nets. In *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 203–212, 2013.
- [21] Guilhem Jaber and Andrzej S. Murawski. Complete trace models of state and control. In Nobuko Yoshida, editor, *Programming Languages and Systems*, pages 348–374, Cham, 2021. Springer International Publishing.
- [22] Marcin Jurdziński and Ranko Lazić. Alternating automata on data trees and XPath satisfiability. *ACM Trans. Comput. Logic*, 12(3), May 2011.
- [23] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.
- [24] Joseph B. Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi’s conjecture. *Transactions of the American Mathematical Society*, 95(2):210–225, 1960.
- [25] Roland Meyer. On boundedness in depth in the π -calculus. In Giorgio Ausiello, Juhani Karhumäki, Giancarlo Mauri, and Luke Ong, editors, *Fifth Ifip International Conference On Theoretical Computer Science – Tcs 2008*, pages 477–489, Boston, MA, 2008. Springer US.
- [26] Torben Æ. Mogensen. A comparison of well-quasi orders on trees. *Electronic Proceedings in Theoretical Computer Science*, 129:30–40, September 2013.
- [27] Crispin ST. J. A. Nash-Williams. *On Well-Quasi-Ordering Finite Trees*, pages 329–331. Birkhäuser Boston, Boston, MA, 1987.
- [28] Frank Neven, Thomas Schwentick, and Victor Vianu. Towards regular languages over infinite alphabets. In *Proceedings of the 26th International Symposium on Mathematical Foundations of Computer Science*, MFCS ’01, page 560–572, Berlin, Heidelberg, 2001. Springer-Verlag.
- [29] Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Logic*, 5(3):403–435, July 2004.

- [30] Elisabeth Pelz. Closure properties of deterministic Petri nets. In Franz J. Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS 87*, pages 371–382, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.
- [31] James L. Peterson. Petri nets. *ACM Comput. Surv.*, 9(3):223–252, September 1977.
- [32] Sylvain Schmitz. *Algorithmic Complexity of Well-Quasi-Orders*. Habilitation à diriger des recherches, École normale supérieure Paris-Saclay, November 2017.
- [33] Sylvain Schmitz and Philippe Schnoebelen. Multiply-recursive upper bounds with Higman’s lemma. In Luca Aceto, Monika Henzinger, and Jiří Sgall, editors, *Automata, Languages and Programming*, pages 441–452, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [34] Sylvain Schmitz and Philippe Schnoebelen. Algorithmic Aspects of WQO Theory. Lecture Notes, August 2012.
- [35] Brian M. Scott. Are the trees well-quasi-ordered by induced subgraphs? Mathematics Stack Exchange, <https://math.stackexchange.com/q/89834>: (version: 2011-12-09).
- [36] Luc Segoufin. Automata and logics for words and trees over an infinite alphabet. In Zoltán Ésik, editor, *Computer Science Logic*, pages 41–57, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- [37] Géraud Sénizergues. $L(A)=L(B)$? Decidability results from complete formal systems. *Theoretical Computer Science*, 251(1):1–166, 2001.

Index

2 counter machine, 53

FOSC, 59

HOSC, 59

$\leq_C$, 10, 11

$\leq_M$, 9

$\leq_T$, 12

$m_k(\mathcal{T})$, 34

admissible, 41

basis, 14

complement, 42

contextual

 approximation, 61

 equivalence, 61

coverability, 14

cr-free, 61

effective pred-basis, 15

image finite, 15

memory graph, 34

NDCMA

 bounded, 30

 flexible, 17

 rigid, 29

nested data set, 16

order

 product, 8

 quasi, 6

 subword, 8

 well-quasi, 6

P-view, 69

Petri net, 50

 reset, 52

post-effective, 15

projection language, 50

termination, 14

Turing machine, 42

upward-closure, 14

well-structured transition system, 13