EPTCS 417

Proceedings of the

Fourteenth and Fifteenth International Workshop on

Graph Computation Models

Leicester, UK and Enschede, the Netherlands, 18 July 2023 and 9 July 2024

Edited by: Jörg Endrullis, Dominik Grzelak, Tobias Heindel and Jens Kosiol

Published: 26th March 2025
DOI: 10.4204/EPTCS.417
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

This volume contains the post-proceedings of the Fourteenth and the Fifteenth International Workshops on Graph Computation Models (GCM 2023 and 2024). The workshops took place in Leicester, UK on 18th July 2023 and Enschede, the Netherlands on 9th July 2024, in each case as part of STAF (Software Technologies: Applications and Foundations).

Graphs are common mathematical structures which are visual and intuitive. They constitute a natural and seamless way for system modeling in science, engineering and beyond, including computer science, life sciences, business processes, etc. Graph computation models constitute a class of very high-level models where graphs are first-class citizens. They generalize classical computation models based on strings or trees, such as Chomsky grammars or term rewrite systems. Their mathematical foundation, in addition to their visual nature, facilitates specification, validation and analysis of complex systems. A variety of computation models have been developed using graphs and rule-based graph transformation. These models include features of programming languages and systems, paradigms for software development, concurrent calculi, local computations and distributed algorithms, and biological and chemical computations. The International Workshop on Graph Computation Models aims at bringing together researchers interested in all aspects of computation models based on graphs and graph transformation. It promotes the cross-fertilizing exchange of ideas and experiences among young and senior researchers from different communities who are interested in the foundations, applications, and implementations of graph computation models and related areas.

Previous editions of the GCM series were held in Natal, Brazil (GCM 2006), in Leicester, UK (GCM 2008), in Enschede, The Netherlands (GCM 2010), in Bremen, Germany (GCM 2012), in York, UK (GCM 2014), in L'Aquila, Italy (GCM 2015), in Wien, Austria (GCM 2016), in Marburg, Germany (GCM 2017), in Toulouse, France (GCM 2018), in Eindhoven, The Netherlands (GCM 2019), online (GCM 2020 and GCM 2021) and in Nantes, France (GCM 2022).

GCM 2024

This post-proceedings volume contains revised versions of six selected papers presented at GCM 2024. All submissions were subject to careful refereeing (at least two reviews per paper), and span a wide range of topics from the foundations and applications of graph computation models.

We would like to thank everyone who contributed to the success of this volume, including our contributing authors, as well as our Programme Committee, who provided several valuable insights throughout the selection process.

Jörg Endrullis and Dominik Grzelak
Program Committee Co-Chairs
Amsterdam (Netherlands) and Dresden (Germany), August 2024

J. Endrullis, D. Grzelak, T. Heindel and J. Kosiol (Eds):
GCM 2023 and GCM 2024
EPTCS 417, 2025, pp. ii–iv, doi:10.4204/EPTCS.417.0

GCM 2024 Program Committee

- Andrea Corradini (Università di Pisa, Italy)
- Jörg Endrullis (Vrije Universiteit Amsterdam, Netherlands)
- Dominik Grzelak (Technische Universität Dresden, Germany)
- Reiko Heckel (University of Leicester, UK)
- Leen Lambers (Brandenburgische Technische Universität Cottbus-Senftenberg, Germany)
- Marino Miculan (University of Udine, Italy)
- Elvira Pino (Universitat Politècnica de Catalunya, Spain)
- Michele Sevegnani (University of Glasgow, UK)
- Vadim Zaytsev (University of Twente, Netherlands)

GCM 2023

This post-proceedings volume contains revised versions of three selected papers presented at GCM 2023. All submissions were subject to careful refereeing (at least three reviews per paper), and span a wide range of topics from the foundations and applications of graph computation models.

We would like to thank everyone who contributed to the success of this volume, including our contributing authors, as well as our Program Committee and additional reviewers, who provided several valuable insights throughout the selection process, and Rob van Glabbeek for all the technical support.

Tobias Heindel and Jens Kosiol
Program Committee Co-Chairs
Berlin (Germany) and Marburg (Germany), September 2024

GCM 2023 Program Committee

- Andrea Corradini (Università di Pisa, Italy)
- Rachid Echahed (CNRS and University of Grenoble, France)
- Fabio Gadducci (Università di Pisa, Italy)
- Reiko Heckel (University of Leicester, UK)
- Tobias Heindel (Heliax Technologies GmbH, Berlin, Germany)
- Berthold Hoffmann (Universität Bremen, Germany)
- Barbara König (University of Duisburg-Essen, Germany)
- Jens Kosiol (Philipps-Universität Marburg and Universität Kassel, Germany)
- Leen Lambers (Brandenburgische Technische Universität Cottbus-Senftenberg, Germany)
- Marino Miculan (University of Udine, Italy)
- Mark Minas (Universität der Bundeswehr München, Germany)
- Detlef Plump (University of York, UK)
- Sven Schneider (Hasso-Plattner-Institut, Universität Potsdam, Germany)

Additional Reviewers

- Nicolas Behr (IRIF, Université Paris Cité, France)
- Davide Castelnovo (University of Udine, Italy)
- Federico Vastarini (University of York, UK)

An Encoding of Interaction Nets in OCaml

Nikolaus Huber
Uppsala University
nikolaus.huber@it.uu.se

Wang Yi
Uppsala University
wang.yi@it.uu.se

Interaction nets constitute a visual programming language grounded in graph transformation. Owing to their distinctive properties, they inherently facilitate parallelism in the rewriting step. This paper showcases a simple and concise approach to encoding interaction nets within the programming language OCaml, emphasising correctness guarantees. To achieve this objective, we encode not only the interaction net primitives, but also Lafont's original type system.

1 Introduction

Interaction nets, introduced by Lafont in [12], are a model of computation based on graph rewriting. They have, among other things, been used as a basis for optimal [3, 13] and efficient implementations [14] of the λ-calculus. Interaction nets offer a number of desirable properties, such as locality of reduction, strong confluence, and Turing completeness. Owing to the locality of reduction, highly parallel implementations can be devised, while the confluence guarantees determinism. They have been studied both as a model for the dynamics of computation and also as a programming language itself. However, only a few implementations are still actively maintained.

In this paper, we study a method of encoding interaction nets into the general purpose programming language OCaml. Unlike previous efforts, our focus is not on the efficiency of reduction, but rather on embedding Lafont's original type system to provide stronger correctness guarantees. By utilising advanced type system features of the host language, we can statically enforce Lafont's typing discipline without relying on dynamic checks.

To the best of our knowledge, this is the first encoding of interaction nets in OCaml, and the first attempt at embedding Lafont's type system into the type system of another language. Given OCaml's recent addition of native support for expressing parallelism, we therefore also explore whether interaction nets can serve as a general implementation scheme for parallel algorithms in the language.

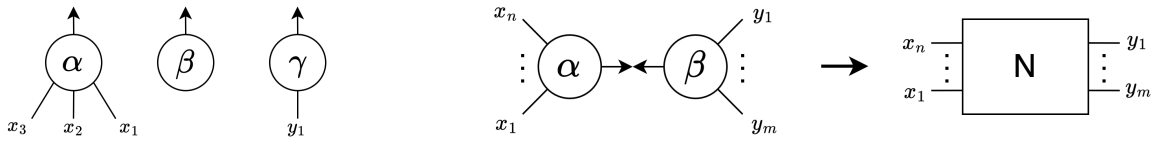
The structure of this paper is as follows: In Section 2 we give an overview of the model of interaction nets together with the graphical notation we will be using throughout the paper, followed by a brief introduction to OCaml and an overview of previous work. In Section 3 we illustrate our method of encoding interaction nets into OCaml by way of showing a concrete example. In Section 4 we look at the runtime behaviour of different encoded nets, and investigate if they can make use of OCaml's support for parallelism. Finally, we give directions for future work and conclude in Section 5.

2 Preliminaries

2.1 Interaction nets

We briefly recall the basic notions of interaction nets. For a more complete presentation, the interested reader is referred to [12]. Interaction nets consist of the following elements:

J. Endrullis, D. Grzelak, T. Heindel and J. Kosiol (Eds):
GCM 2023 and GCM 2024
EPTCS 417, 2025, pp. 1–16, doi:10.4204/EPTCS.417.1



(a) Example agents

(b) Example rule

Figure 1: Graphical notations for agents and rules

- A set Σ of *symbols*. These symbols are used as labels for nodes, which are referred to as *agents*. Each agent has a fixed set of *ports*, through which they connect with other agents. An agent has exactly one distinguished *principle port*, and a (possibly empty) set of *auxiliary ports*. The number of auxiliary ports is fixed for each symbol. We assume the existence of a function $ar : \Sigma \to \mathbb{N}$, such that, for all $\alpha \in \Sigma$, $ar(\alpha)$ is the number of auxiliary ports associated with the symbol α. Figure 1a illustrates the usual graphical notation for agents of three different symbols α, β, and γ. The principle port is depicted as an arrow, auxiliary ports are numbered clockwise from the principle port. In the given example $ar(\alpha) = 3$, $ar(\beta) = 0$, and $ar(\gamma) = 1$.
- A net N is an undirected graph built from agents over the set Σ. Each edge of the graph connects two agent ports together, such that there is only one edge per port. A port that is not connected is called a *free port*. The set of free ports of a net is called its *interface*.
- A set $\mathscr{R}$ of *rules*. A pair of agents $(\alpha, \beta) \in \Sigma \times \Sigma$ is called an *active pair* if they are connected through their principle ports (this is related to the concept of a *redex* in term rewriting). The application of an interaction rule $((\alpha, \beta) \to N) \in \mathscr{R}$ replaces an active pair by a net N in such a way that the interface of N coincides exactly with the auxiliary ports of α and β. Intuitively, this means, that all auxiliary connections of the active pair must be connected to the new net, and no new free ports can be introduced in the process. Figure 1b shows an example of the graphical notation for rules.

Rewriting happens through the application of interaction rules to active pairs. Since every agent only has one distinct principle port, the rewriting process is local, and two different active pairs can be rewritten in parallel. Some additional requirements are necessary to guarantee determinism: For each pair (α, β) there can only be at most one rule in $\mathscr{R}$, and if there is no rule for a specific active pair it cannot be further reduced. Since there is no notion of orientation, rules are symmetric, meaning that $(\alpha, \beta) \to N$ and $(\beta, \alpha) \to N$ describe the same rule. With these restrictions in place it can be shown [12], that the reduction sequence is strongly confluent.

2.2 Typed interaction nets

Already in the original paper [12], Lafont remarks that not all possible connections between agents have a valid semantic interpretation. In the next section, we will introduce agents representing boolean and integer values, as well as agents representing functions over those values. A function agent expecting a particular type at its principle port cannot be reduced when paired with an agent of the wrong type. Therefore, Lafont introduced a rudimentary type system, where each port of an agent is assigned both a *value type*, and a *polarity*. Value types can include integers, booleans, floats, lists, etc.

The polarity encodes if a port is meant to be an input or an output. An example can be seen in Figure 2. It is up to convention if inputs receive positive, or negative polarity, as long as the assignment

is consistent throughout. In this paper, we will follow the same convention as in [12], where inputs have negative, and outputs positive polarity.

Only ports of the same type, but opposite polarity, can be connected to each other. Rules are well typed if their left-hand side is well typed (i.e., the principle ports of the active pair have the same type and opposite polarity), and if the net on the right-hand side is well typed, taking into account the interface types provided by the auxiliary connections of the active pair.

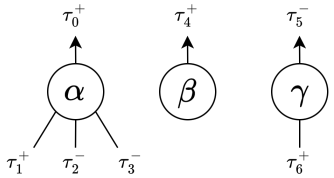


Figure 2: Examples of typed agents

2.3 OCaml

OCaml is an industrial-strength (primarily functional) programming language which arose as an extension of the Caml dialect of the ML language family. While it is a general-purpose language, it has a strong background in theorem proving (e.g., Coq [1]), static analysis (e.g., Frama-C [11]), and formal methods software. Interestingly, even though it is a language used in industry, it only recently added native support for expressing parallel evaluation [20].

The basic unit of parallelism in OCaml is called *domains*. A single domain is created at the start of a program, and more domains can be spawned later on. Domains map directly to operating system threads, and are therefore costly to create and tear down. It is thus recommended to not spawn more domains than cores available on the local processor. While the standard library of OCaml provides the `Domain` module, it only offers low-level primitives for managing domains. This opens the possibility of providing higher-level libraries to be developed outside the core compiler. Different such libraries have been developed, all of them with different design trade-offs and different APIs. In order not to be too restricted by the choice of library, we will show our encoding method against a simplified API, which can be provided by different libraries:

```
type pool
type 'a promise
type 'a resolver
val resolve : 'a resolver -> 'a -> unit
val await : 'a promise -> 'a
val block : 'a promise -> 'a
val make_future : unit -> 'a promise * 'a resolver
val create_pool : int -> pool
val run_async : pool -> (unit -> unit) -> unit
```

Type `pool` describes the pool of worker threads, it is created by a call to `create_pool` with the number of desired workers as an argument. Types `'a promise` and `'a resolver` are two ends of a simple communication primitive. They are always created together by a call to `make_future`. A value

of type `'a promise` acts as a placeholder for a value of type `'a`, which will later be supplied through the corresponding `resolver` by calling the `resolve` function. Function `await` takes a promise and returns its value once it is resolved. Function `block` does the same. However, it blocks the currently running thread until the promise is resolved. Finally, the function `run_async` takes a pool and a continuation, and puts that continuation into the work-queue of the pool for asynchronous execution. Such an API can be provided by different already existing libraries, the one we chose for this paper is called Moonpool [2]. For those unfamiliar with the OCaml language, we give a short overview of the syntax in appendix A.

2.4 Previous work and contributions

A number of different evaluators/compilers for interaction nets have been developed. Some have built upon each other, such as AMINE [17], PIN [4], INET [5], and amineLight [15]. A comparison of these is given in [18], and the most recent and still actively maintained version of this lineage is available in the inpla project [19].

Due to the inherent possibility of parallel reduction, running interaction nets on GPUs was investigated in [8]. The code for the project is still available [7]. However, it has not been updated in 13 years and needs all rules encoded as custom CUDA kernels.

Encoding interaction nets in a functional programming language was done in [10], where interaction nets were embedded into Concurrent Haskell [16]. The code for this evaluator is available [9], but it also has not been updated since 2015, and requires a rather old version of Haskell to still compile.

Our method is closest to the embedding into Haskell. However, we have taken multiple different design choices. In the Haskell encoding, the arity constraint for each label is not enforced by the type system, as ports are encoded as a list of directed references to other agents. In our encoding, not all connections are implemented as references, and utilising a dedicated constructor for each agent the type system guarantees the arity constraints. The Haskell encoding uses polarity to figure out the direction for each reference, i.e., which side will provide the reference, and which side waits until the reference is provided. It relies on a heuristic to figure out the polarity of each port. In this paper, we bypass the inference of polarities, and instead assume that they are given as part of the agent definitions. In the Haskell encoding, polarities are carried around as concrete values, and are dynamically checked. We encode polarities in the type system of OCaml directly, therefore preventing the construction of incorrect nets already during compile time, and removing the need for dynamic checks.

3 Encoding

In this section, we showcase the encoding of the different constituent parts of interaction nets into OCaml. We start by defining a dedicated type for agents, then show how we encode rule application. After introducing agent attributes, we then show how we embedded Lafont's original type system into the type system of OCaml.

3.1 Encoding agents

There are many ways of how agents might be encoded within a given language. In the interaction net to C compiler presented in [5], agents are encoded as a structure, where the first field relates to the label of the respective agent, while the second encodes the ports as an array of references to other agents. The encoding of interaction nets in Haskell presented in [10] uses a similar approach, where each agent is described as a record with a field for the label and a field for the list of connections to other agents.

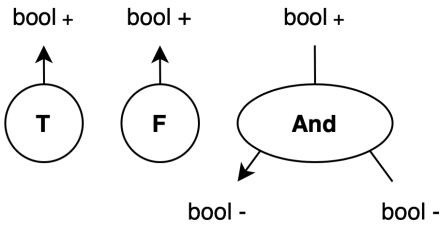


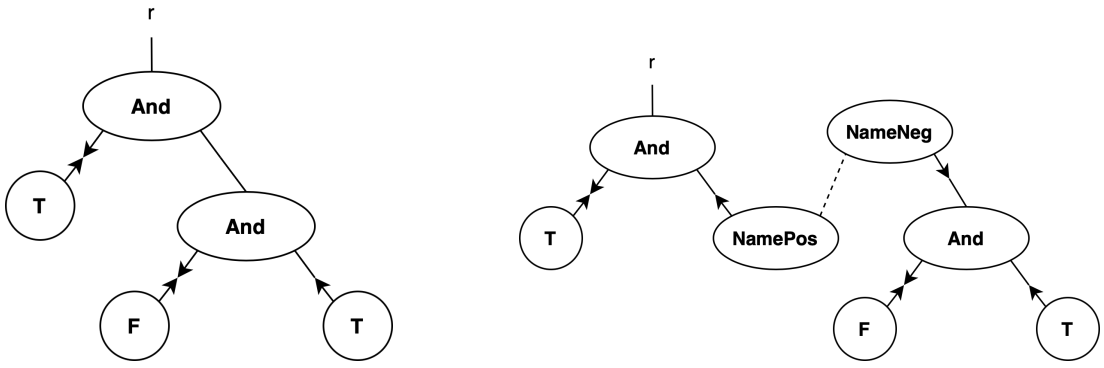
Figure 3: Boolean agents

These connections make use of mutable variables, as introduced by Concurrent Haskell [16]. Indeed, the way that mutual connections of agents are encoded seems to be one of the most important differences between different interaction net evaluators. In his PhD thesis [18], Sato gives a nice overview of these different encodings. According to his nomenclature, our encoding method follows the principle of *single link encoding*, which seems to also be the basis of the Haskell encoding.

For a given interaction net system with a label set Σ, we create a *variant data type* with one constructor case per symbol. As a simple example, we will start by encoding the agents shown in Figure 3:

```
type agent =
    | T
    | F
    | And of agent * agent
```

Each symbol $\alpha \in \Sigma$ is translated to one constructor case in the type `agent`, using α as the label, and carrying $ar(\alpha)$ agents as attributes (encoding the appropriate number of auxiliary ports). When creating a value of type `agent` by using one of these constructors, the resulting value represents the principle port of the respective agent. Every auxiliary port used as an attribute is a connection to the principal port of another agent.



(a) Using common connection

(b) Using name nodes

Figure 4: Connection between auxiliary ports

Nets are created by simply applying constructors recursively to each other. However, encoding a net in this way does not allow for two auxiliary ports to be connected directly, so the net in Figure 4a could not be encoded in this way. To allow the mutual connection of auxiliary ports, we break the link and

create two agents with new symbols `NamePos` and `NameNeg` (see Figure 4b). We utilise the information about the polarity of the connection in order to decide which port needs to be connected to which name agent. We can therefore extend our `agent` type accordingly:

```
type agent =
    | T
    | F
    | And of agent * agent
    | NamePos of agent promise
    | NameNeg of agent resolver
```

We use the same mechanism for creating connections for the interface of the initial network (i.e., the network to be rewritten). We also introduce a function to create these pairs of name agents:

```
let new_name () =
    let promise, resolver = make_future () in
    NamePos promise, NameNeg resolver
```

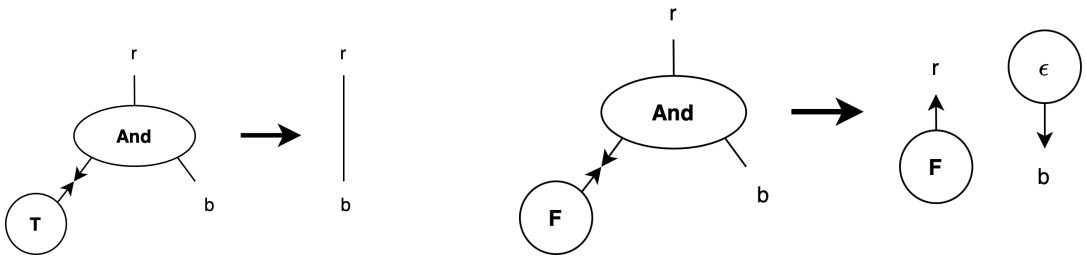
3.2 Encoding rule application

Apart from agents, we also need to encode rules. All rules are collected and translated together into a function `apply_rule`, which internally uses pattern matching to match the different active pairs. So for the two rules shown in Figure 5 we get:

```
let rec apply_rule a1 a2 = match a1, a2 with
    (* using an or-pattern to list both orientations *)
    | T, And (r, b)
    | And (r, b), T          -> b -><- r
    | F, And (r, b)
    | And (r, b), F          -> ignore b; F -><- r
    | NamePos v, a
    | a, NamePos v           -> await v -><- a
    | NameNeg v, a
    | a, NameNeg v           -> resolve v a
    (* match cases must be exhaustive in OCaml *)
    | _, _                   -> failwith "No rule for this pair"

(* custom infix operator to run rewriting asynchronously in thread pool *)
and ( -><- ) a1 a2 = run_async pool (fun _ -> apply_rule a1 a2)
```

Since all rules are symmetric in the constituents of the active pair, we have to list both orientations. OCaml has a shorthand syntax for when two cases have the same right-hand side, as long as we name the attributes of matched constructors the same. The functions `apply_rule` and `-><-` are mutually recursive, the latter uses OCaml's support for custom infix operators and just puts the application of `apply_rule` into the pool of threads to run asynchronously with all other rewritings. While this is not



(a) Rule T →← And

(b) Rule F →← And

Figure 5: Rules for conjunction

strictly necessary, it simplifies the code in the `apply_rule` method considerably. We assume that the variable `pool` has been globally defined at the start of the program. To satisfy the exhaustiveness of the pattern match, we need to include a catch-all case (`_, _`), which just fails with an error message. As we will see in Section 3.5, once we introduce Lafont's typing discipline, this catch-all case will become unnecessary.

The careful reader will have realised that we have not introduced the agent with label ε as shown in Figure 5b. This is a special agent, usually referred to as the *delete agent*, which just recursively deletes the net connected to it. As OCaml is a managed language, we can leave this deletion process to the garbage collector. However, the compiler will complain if we name a port on the left-hand side of a case and not use it on the right-hand side. We therefore use the OCaml function `ignore` to mark that we do not care about a particular value. Another option would have been to use the special `_` pattern in order not to give a name to the unused port:

```
...
    | F, And (r, _)
    | And (r, _), F    -> F ->< r
...
```

3.3 Agent attributes

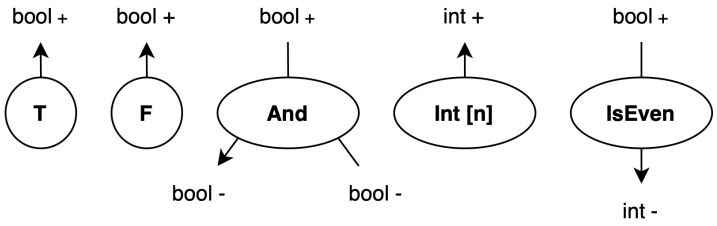


Figure 6: Boolean agents extended with integer agents

Representing booleans with two different labels works fine. However, with numbers such a unary encoding becomes already quite tedious and inefficient. Therefore, we extend our definition of agents, allowing them to carry attributes, i.e., values. As shown in [5], this does not contradict the strong confluence of the reduction. Figure 6 shows an extension of our previous label set with `Int` agents, carrying

a value of their namesake, as well as an agent `IsEven`. Encoding agents with attributes in OCaml is straightforward, as we can just extend the attributes of the respective constructor:

```
type agent =
    | Int of int
    | IsEven of agent
    | T
    | F
    | And of agent * agent
    | NamePos of agent promise
    | NameNeg of agent resolver
```

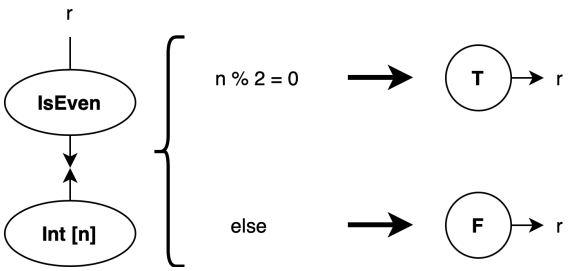


Figure 7: Rule Int[n] →← IsEven

We can use the attributes of the agents on the left-hand side of a rule to define guarded right-hand sides, as shown in Figure 7. Encoding such rules in `apply_rule` is easy, as OCaml allows match cases to be guarded by a boolean expression:

```
let rec apply_rule a1 a2 = match a1, a2 with
    ...
    | IsEven r, Int n
    | Int n, IsEven r when n mod 2 = 0    -> T -><- r
    | IsEven r, Int _
    | Int _, IsEven r                     -> F -><- r
    ...
```

3.4 Type compatibility

With the introduction of integers into our example, we now also have to consider the types of ports. Indeed, the way we have defined agents through an ordinary variant type does not enforce this, so the OCaml compiler will happily accept `And (r, Int 0)` as a valid expression, even though `Int 0` is not a boolean. We could, of course, try and include a deep embedding of the type of an agent by doing something similar to the following:

```
type ty =
    | Int
    | Bool
```

```
type agent =
    | Int of ty * int
    | IsEven of ty * agent
    | T of ty
    | F of ty
    | And of ty * agent * agent
    | ...
```

However, encoding the type in this way incurs an overhead both in execution time and code conciseness, as we would need to include runtime checks at appropriate positions to check for type compatibility.

It would be nicer, if we could track the type of an agent in OCaml's type system directly, in a way that would prevent the construction of such faulty nets in the first place. Luckily, OCaml offers exactly this in the form of *generalised algebraic data types* (GADTs). A GADT is similar to a parameterised variant type, but it allows expressing constraints on the type variable for each constructor case individually. For our example, the agent type can be defined in the following way:

```
type _ agent =
  | Int : int -> int agent
  | IsEven : bool agent -> int agent
  | T : bool agent
  | F : bool agent
  | And : bool agent * bool agent -> bool agent
  | NamePos : 'a agent promise -> 'a agent
  | NameNeg : 'a agent resolver -> 'a agent
```

The `:` after the constructor label indicates the definition of a GADT. The `_` before `agent` is an *anonymous type variable*, as it does not show up in the definition of the type `agent` itself. However, for each constructor case this type variable can be constrained to a concrete type, thereby limiting the allowed agents at specific auxiliary ports. For example, the constructor `And` expresses, that both auxiliary agent connections must be of type `bool`, the principle port has type `bool` as well (which, by lucky syntactic coincidence, follows after the `->`). With this in place, the OCaml compiler will now reject `And (r, Int 0)` as a valid net.

The type checker usually needs additional information when GADTs are paired with recursive functions (such as `apply_rule`). In general, if a recursive function uses pattern matching, the type checker fails to unify the type variable with different types in the different cases. It also does not allow for a recursive call to use different type variables than the outer call. This can be circumvented by providing explicit type annotations that mark the function as polymorphic in the type parameter of the types of its input:

```
let rec apply_rule : type a. a agent -> a agent -> unit =
    fun a1 a2 -> ...

and ( ->><- ) : type a. a agent -> a agent -> unit  =
    fun a1 a2 -> ...
```

Intuitively, the annotation `type a` can be read as a for-all quantifier.

3.5 Polarity

Type compatibility between ports is only one part of Lafont's type system. It does not prevent us from creating active pairs that do not fit together in terms of their polarity, so currently, the compiler will accept `Int 0 ->><- Int 1` as valid. To encode the polarity, we can introduce a second anonymous type variable for the definition of the polarity type. If we, for now, assume that there are defined types `pos` and `neg`, we can express the agent type in the following way:

```
type (_, _) agent =
  | Int : int -> (int, pos) agent
  | IsEven : (bool, neg) agent -> (int, neg) agent
  | T : (bool, pos) agent
  | F : (bool, pos) agent
  | And : (bool, neg) agent * (bool, pos) agent -> (bool, neg) agent
  | NamePos : ('a, pos) agent promise -> ('a, pos) agent
  | NameNeg : ('a, pos) agent resolver -> ('a, neg) agent
```

The astute reader will have surely realised, that all auxiliary ports in the above definition have opposite polarity to what they had in their graphical depiction in Figure 6. This change in polarity is due to the fact that we need to think in terms of the polarity of the agents that will be connected to these ports, therefore we need to invert them. With these types in place, we can refine the definition of `apply_rule`:

```
let rec apply_rule : type a. (a, pos) agent -> (a, neg) agent -> unit =
  fun a1 a2 -> match a1, a2 with
  | T, And (r, b)                    -> b ->><- r
  | F, And (r, b)                    -> ignore b; F ->><- r
  | Int n, IsEven r when n mod 2 = 0 -> T ->><- r
  | Int _, IsEven r                  -> F ->><- r
  | T, If (r, t, _)                  -> t ->><- r
  | F, If (r, _, e)                  -> e ->><- r
  | NamePos v, a                     -> await v ->><- a
  | a, NameNeg v                     -> resolve v a

and ( ->><- ) : type a. (a, pos) agent -> (a, neg) agent -> unit =
  fun a1 a2 -> run_async pool (fun _ -> apply_rule a1 a2)
```

Interactions can only happen between agents that have the same type but opposite polarity. The choice of having the first agent be positive, and the second negative is arbitrary, and we could have introduced them just as well the other way around.

With these type annotations in place, the OCaml compiler will now also inform us, that the catch-all case is superfluous, as we have accounted for all possible active pair patterns. It also means that only one orientation of the agents of an active pair is permitted in the pattern match any more, so we can shorten the code quite a bit.

In it worth noticing, that the OCaml compiler performs not only exhaustiveness checks (see A), but also checks for overlaps in the case patterns. The order of cases matters (e.g., in the above example it is important that the case with the `when` guard is listed first), so later case patterns can generalise prior ones, but if two cases are overlapping completely, the OCaml compiler would complain. This further increases confidence, that the encoding is correct.

It remains to define the two types `pos` and `neg`. In principle, any two types that cannot be unified would work, so we could make `pos` a synonym for `int` and `neg` a synonym for `float`, for example. However, since we never create values of these types (they are only used as *tags* in the type definition), the most idiomatic choice is an *uninhabited* or *empty type*. OCaml allows the definition of such a type as a variant without constructors:

```
type pos = |
type neg = |
```

Since they are defined as two separate types, the type checker can never unify them. We could have, of course, encoded the polarity dynamically as another attribute of all agent constructors again, and put checks at relevant places (e.g., at agent construction and `apply_rule` applications). This would have come at an additional cost in both execution time and code size again. By tracking the polarity in the type system we incur neither of these, since types are removed by the compiler after type checking, and the compiler will reject expressions such as `Int 0 -><- Int 1` already at compile time.

4 Benchmarks

While the main goal of this paper is to express a clean, concise, and correct encoding of interaction nets into OCaml, it is of course interesting to look at the runtime behaviour of the encoded nets as well. To this end, we have encoded three example nets, taken from the inpla project [19]. Specifically, we have encoded nets for calculating Fibonacci numbers, and sorting lists of integers with Quicksort and Mergesort. Our code is available online [6].

All examples were run on a 3.70 GHz Intel Core i9 processor with 10 cores and 2 hardware-threads per core. For each measurement we average over 100 runs. The first question is, of course, if the interaction net encoding can make use of the available cores. Figure 8 shows the relative speed-up ratios when run on pools of different sizes. It is important to notice that the library we are using (Moonpool [2]) works in terms of thread pools, not domain pools. There is always exactly one domain pool created at the start of the program (with the recommended number of domains, i.e., the number of cores/hardware-threads available on the local processor), the library then creates pools of worker threads which share these domains. There are a couple of interesting observations:

Fibonacci scales extremely well with regard to the size of the provided pool, while Quicksort's and Mergesort's performances only marginally increase, before they deteriorate for higher thread counts.

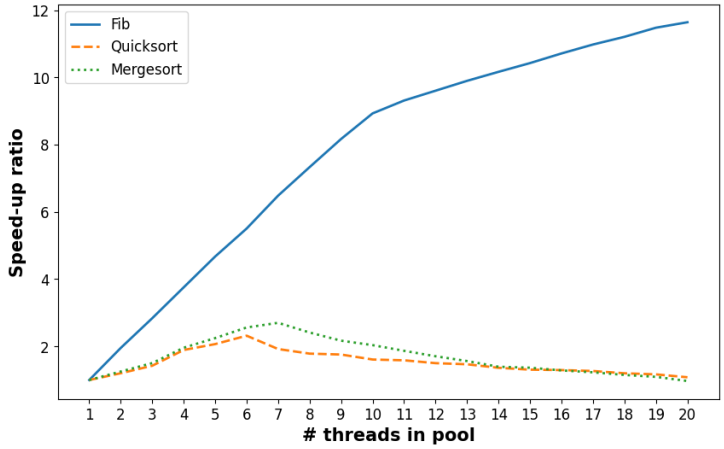


Figure 8: Relative speed-up factors for different pool sizes

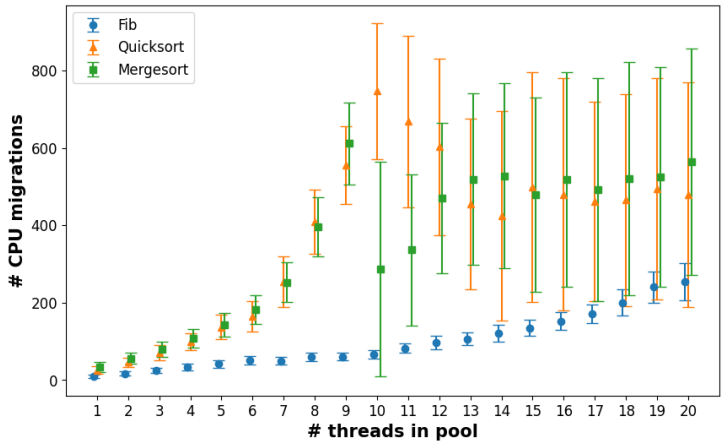
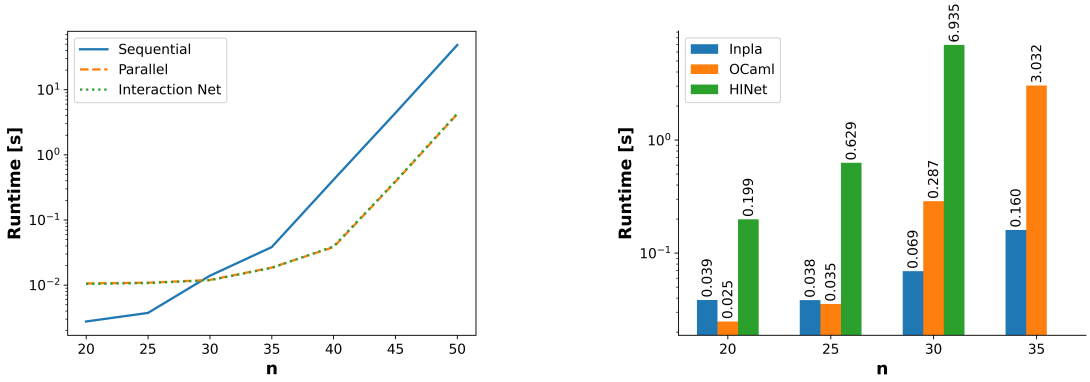


Figure 9: Number of CPU migrations for different pool sizes



(a) Runtime compared to handwritten OCaml

(b) Tool comparison

Figure 10: Fibonacci benchmark

There are several possible reasons for this. As a first indicator, we can look at the average amount of CPU migrations (threads moving from one CPU to another) for different thread counts, as shown in Figure 9. The average number of CPU migrations for Fibonacci increases only slightly for higher thread counts, while showing exponential increases for both sorting algorithms until the number of physical cores is reached. We can also see that the average number of CPU migrations for both sorting algorithms fluctuates significantly for higher thread counts, as indicated by the standard deviations. While this gives a possible explanation for the lack of relative speed-up, further investigation is needed to look at other performance indicators such as cache misses and garbage collection behaviour.

The speed-up ratios are relative to the performance of the sequential execution, they do not allow to reason about absolute performance. We have therefore also compared the performance of the Fibonacci encoding against the sequential and a handwritten parallel OCaml implementation. The results are shown in Figure 10a. Parallelism incurs a price in terms of runtime, so for small inputs (≤ 20) we revert to the sequential algorithm for both the interaction net encoding and the parallel implementation. This shows another advantage of embedding interaction nets in a host language, as such tricks can easily be encoded.

It is interesting to notice that the interaction net encoding is fairly on par with the handwritten parallel code. This suggests, that at least for some algorithms, interaction nets can be a valid candidate for implementing parallel algorithms with fine-grained parallelism in OCaml.

For completeness, Figure 10b compares the runtime of the Fibonacci net for different inputs n in our encoding with two other interaction net evaluators. This comparison is not entirely fair, as they use vastly different strategies for running the provided net. Inpla [19] compiles interaction nets into bytecode for a low-level virtual machine dedicated to the execution of interaction nets. HINet [9] encodes interaction nets in Haskell dynamically, and then interprets these. It is therefore not surprising, that it is slower than the two other approaches. For the last test input, HINet did not produce an output within 10 minutes, and was therefore stopped.

5 Conclusion

In this paper, we presented a method of encoding interaction nets in the programming language OCaml. Our main focus was on an encoding of both the interaction net primitives (agents, nets, rules), as well as Lafont's type system.

Compared to previous work, our encoding offers stronger guarantees of correctness. By defining a variant type with a constructor for each symbol, the type system guarantees the arity constraint and types of agent attributes. Using a GADT we can express both the value type and polarity of each port so that port compatibility is already checked at compile-time. Furthermore, it allows the OCaml type checker to prove the exhaustiveness of the rewriting step (i.e., that all possible active-pair patterns are accounted for). Since the encoding is purely done on the level of types, no additional runtime cost is incurred.

Experiments indicate that the encoding of certain algorithms can make use of the inherent parallelism. Others seem to be constrained by the scheduling of individual threads on the available cores. More investigation is needed regarding different performance counters and optimal scheduling of the rewriting process.

Another direction of future work relates to the choice of library to provide the primitives. As described in the introduction, the idea behind OCaml's low-level primitives in the standard library is that different (opinionated) higher-level libraries can be developed on top of it. The library we have used in this paper is only one possible option, and it would be interesting to see, if the choice of library has a significant impact on the runtime behaviour of the net evaluation.

In this paper, we demonstrated how to encode interaction nets in OCaml manually. To allow for better comparisons with other tools, a compiler could be developed, taking as input a language similar to that of previous interaction net evaluators. OCaml offers an inbuilt system for language extension, therefore by developing an extension for interaction nets, interaction net algorithms could be embedded directly into a surrounding program, while automatically using parallel evaluation if run on a multicore platform.

Finally, we want to remark, that while our focus here was on an encoding in OCaml, our method is general enough to be used for any language that offers GADTs and concurrency.

Acknowledgements This work is partially supported by the ERC CUSTOMER project.

References

[1] Yves Bertot & Pierre Castran (2010): *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*, 1st edition. Springer Publishing Company, Incorporated, doi:10.1007/978-3-662-07964-5.

[2] Simon Cruanes (2024): *Moonpool*. Available at `https://github.com/c-cube/moonpool`. Version 0.6.

[3] Georges Gonthier, Martín Abadi & Jean-Jacques Lévy (1992): *The geometry of optimal lambda reduction*. In: *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, Association for Computing Machinery, New York, NY, USA, p. 15–26, doi:10.1145/143165.143172.

[4] Abubakar Hassan, Ian Mackie & Shinya Sato (2008): *Interaction nets: programming language design and implementation*. *ECEASST* 10, doi:10.14279/tuj.eceasst.10.156.

[5] Abubakar Hassan, Ian Mackie & Shinya Sato (2009): *Compilation of Interaction Nets*. *Electronic Notes in Theoretical Computer Science* 253(4), pp. 73–90, doi:10.1016/j.entcs.2009.10.018. Proceedings of the Fifth International Workshop on Computing with Terms and Graphs (TERMGRAPH 2009).

[6] Nikolaus Huber (2024): *An Encoding of Interaction Nets in OCaml - Software Artifact*, doi:10.5281/zenodo.12633477. Available at `https://doi.org/10.5281/zenodo.12633477`.

[7] Eugen Jiresch (2011): *ingpu*. Available at `https://github.com/euschn/ingpu`.

[8] Eugen Jiresch (2014): *Towards a GPU-based implementation of interaction nets*. *Electronic Proceedings in Theoretical Computer Science* 143, p. 41–53, doi:10.4204/eptcs.143.4.

[9] Wolfram Kahl (2015): *HINet - Interaction Nets in Haskell*. Available at `http://www.cas.mcmaster.ca/~kahl/Haskell/HINet/`.

[10] Wolfram Kahl (2015): *A Simple Parallel Implementation of Interaction Nets in Haskell*. *Electronic Proceedings in Theoretical Computer Science* 179, p. 33–47, doi:10.4204/eptcs.179.3.

[11] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles & Boris Yakobowski (2015): *Frama-C: A software analysis perspective*. *Formal aspects of computing* 27(3), pp. 573–609, doi:10.1007/s00165-014-0326-7.

[12] Yves Lafont (1989): *Interaction nets*. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, Association for Computing Machinery, New York, NY, USA, p. 95–108, doi:10.1145/96709.96718.

[13] John Lamping (1989): *An algorithm for optimal lambda calculus reduction*. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, Association for Computing Machinery, New York, NY, USA, p. 16–30, doi:10.1145/96709.96711.

[14] Ian Mackie (1998): *YALE: yet another lambda evaluator based on interaction nets*. In: *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, Association for Computing Machinery, New York, NY, USA, p. 117–128, doi:10.1145/289423.289434.

[15] Ian Mackie & Shinya Sato (2010): *A lightweight abstract machine for interaction nets*. *ECEASST* 29, doi:10.14279/tuj.eceasst.29.416.

[16] Simon Peyton Jones, Andrew Gordon & Sigbjorn Finne (1996): *Concurrent Haskell*. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, Association for Computing Machinery, New York, NY, USA, p. 295–308, doi:10.1145/237721.237794.

[17] Jorge Sousa Pinto (2000): *Sequential and Concurrent Abstract Machines for Interaction Nets*. In Jerzy Tiuryn, editor: *Foundations of Software Science and Computation Structures*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 267–282, doi:10.1007/3-540-46432-8_18.

[18] Shinya Sato (2015): *Design and implementation of a low-level language for interaction nets*. Ph.D. thesis, University of Sussex. Available at `https://sussex.figshare.com/articles/thesis/Design_and_implementation_of_a_low-level_language_for_interaction_nets/23417312`.

[19] Shinya Sato (2023): *inpla*. Available at `https://github.com/inpla/inpla`. Version 0.11.0.

[20] KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman & Anil Madhavapeddy (2020): *Retrofitting parallelism onto OCaml*. *Proc. ACM Program. Lang.* 4(ICFP), doi:10.1145/3408995.

A Overview of OCaml

The basic units of any OCaml program are *expressions*, such as `10 + 25`. Each expression has a *type*, such as int, float, bool, string, etc., and can be bound to a name via a `let` directive. The OCaml compiler uses type inference, so that for most ordinary cases no type annotations are needed. However, a programmer can always express the types of symbols explicitly if desired. The following definitions are equivalent:

```
let x = 10 + 25
let x : int = 10 + 25
```

Anonymous functions can be implemented with the `fun` keyword. As functions are first class values, they can also be bound to a name. Function types are indicated with the `->` symbol. The following definitions are all semantically equivalent:

```
let square = fun x -> x * x
let square : int -> int = fun x -> x * x
let square x = x * x
let square (x : int) : int = x * x
```

Functions are curried, so partial application is possible:

```
let add x y = x + y
let add2 = add 2
let sum = add2 4
```

The core of OCaml's expressive power comes from the ability to define rich data types. The one most used in this paper is called *variant data type* (a generalisation of enumeration and union types):

```
type myType =
    | A of int
    | B of bool
```

The above expression introduces a type `myType`. Values of `myType` are *either* an `A` carrying an integer as an attribute, *or* a `B` carrying a boolean. Variants are constructed by applying their constructor name (e.g., `A` or `B`) to the appropriate number of argument expressions according to the type definitions. Variants can be deconstructed by *pattern matching*:

```
let x : myType = A 1
let output = match x with
    | A n -> if n = 1 then true else false
    | B b -> not b
```

We can see that the value carried by a constructor can be bound to a name on the left-hand side of a case, and then be referred to in the expression on the right-hand side. All cases of a pattern match must return the same type. Matches are also checked for exhaustiveness, so if we change the above example to

```
let x : myType = A 1
let output = match x with
    | A n -> if n = 1 then true else false
```

the compiler will complain:

```
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
B _
```

Variants can be parameterized in terms of other types. For example, the OCaml standard library defines the list type in the following way:

```
type 'a list =
    | Nil
    | Cons of 'a * 'a list
```

The *type parameter* `'a` can be instantiated to any already defined type. Type synonyms can be introduced to make the code more readable:

```
type myList = myType list
let l : myList = Cons (A 1, Cons (B false, Nil))
```

Type constructors are not functions, so their application cannot be partial.

Modelling Privacy Compliance in Cross-border Data Transfers with Bigraphs

Ebtihal Althubiti
Northern Border University, Arar, Saudi Arabia
ebtihal.althubiti@nbu.edu.sa
University of Glasgow, Glasgow, United Kingdom
e.althubiti.1@research.gla.ac.uk

Michele Sevegnani
University of Glasgow, Glasgow, United Kingdom
michele.sevegnani@glasgow.ac.uk

Advancements in information technology have led to the sharing of users' data across borders, raising privacy concerns, particularly when destination countries lack adequate protection measures. Regulations like the European General Data Protection Regulation (GDPR) govern international data transfers, imposing significant fines on companies failing to comply. To achieve compliance, we propose a privacy framework based on Milner's Bigraphical Reactive Systems (BRSs), a formalism modelling spatial and non-spatial relationships between entities. BRSs evolve over time via user-specified rewriting rules, defined algebraically and diagrammatically. In this paper, we rely on diagrammatic notations, enabling adoption by end-users and privacy experts without formal modelling backgrounds. The framework comprises predefined privacy reaction rules modelling GDPR requirements for international data transfers, properties expressed in Computation Tree Logic (CTL) to automatically verify these requirements with a model checker and sorting schemes to statically ensure models are well-formed. We demonstrate the framework's applicability by modelling WhatsApp's privacy policies.

1 Introduction

Transferring data across various jurisdictions worldwide is one of the requirements for advancing digital systems [27]. For example, WhatsApp states that they need to share users' data with Meta's data centres and Facebook's branches around the world to improve their services [33]. Transferring data to different countries can potentially threaten users' privacy as the receiver country may not use sufficient mechanisms to protect the data, *e.g.* encryption techniques [10].

Several regulations have been imposed to restrict such transfers, for example, the European Union (EU) General Data Protection Regulation (GDPR) [15], and the Australian Privacy Principles (APPs) [5]. The main aim of these regulations is to guarantee that the recipient country provides an adequate data protection level compared to the sender country's protection level.

According to the GDPR, there are two main ways to restrict data transfer towards non-EU countries: transferring based on the *adequacy decision* or transferring based on *appropriate safeguards* [9]. The former permits the transfer to specific countries that have an adequate level of data protection as specified by the European Commission. The latter requires safeguards, *e.g. Standard Contractual Clauses (SCCs)* or a certification, if the adequacy decision does not cover the receiver country. Besides these two methods, the GDPR identifies some exceptional cases where transferring data is permitted, *e.g.* if the transfer is important for the public interest. We discuss all these ways further in Section 2.

Organisations must adhere to the above requirements to transfer data from the EU to third countries. Otherwise, they can be at risk of being fined by an EU authority. For example, Facebook has been fined 1.2 billion euro by the Irish Data Protection Authority (IE DPA) for not adhering to the GDPR

J. Endrullis, D. Grzelak, T. Heindel and J. Kosiol (Eds):
GCM 2023 and GCM 2024
EPTCS 417, 2025, pp. 17–38, doi:10.4204/EPTCS.417.2

requirements by transferring personal data from EU countries to the US[1] [7]. Such breaches could occur for many reasons. One is that the regulations are text-based and subject to update, which can create challenges for developers to convert them into technical requirements [17]. Another reason is that tools used by developers or supervisory authorities to check compliance are not automated, *e.g.* Transfer Impact Assessment (TIA) [30, 17]. This motivates the need for an automated approach that enables organisations to ensure their systems comply with the privacy regulations and serve as evidence of their compliance for supervisory authorities.

Formal methods are mathematical techniques that have been used in various domains to solve these issues thanks to their ability to model systems and provide a rigorous analysis of properties of interest such as security and safety [35]. They can also be used as proof of fulfilling these specifications because they enable automated and comprehensive verification processes. To the best of our knowledge, they have yet to be used to prove the systems' compliance with the GDPR requirements for cross-border personal data transfers.

In this paper, we propose a framework based on Milner's Bigraphs [25] to model systems and prove for the first time their compliance to these aspects of privacy regulations. Bigraphs are a universal formalism for the modelling of interacting systems that evolve in their connectivity and space via rewriting rules called *reaction rules*. Compared to other formalisms, reaction rules are user-specified allowing for greater flexibility to model a wide variety of systems. Bigraphs and reaction rules can be specified algebraically or through an equivalent diagram notation. This feature enables system designers to collaborate with privacy experts, *e.g.* privacy lawyers, who might not be expert in formal modelling and verification. Another advantage of using bigraphs is that modelling the flow of data among different jurisdictions requires modelling the spatial relationships between entities, and bigraphs can natively express spatial properties such as containment relation. Additionally, the sorting discipline provides another key advantage by categorising entities and links by means of sorts, ensuring that bigraphs are well-formed and preventing invalid compositions. This adds a layer of structural validation, making the model more robust.

Our approach is described in Fig. 1. The privacy aspects of our framework are predefined while system-specific aspects have to be specified for each application. The framework also provides a set of privacy properties that should be verified, *e.g.* providing safeguards, the validity of them *etc*. These privacy properties are expressed using the Computation Tree Logic (CTL) and can be checked *automatically* using off-the-shelf verification tools such as PRISM [22].

We provide the following contributions:

- We define a bigraph framework to model the GDPR requirements for cross-border data transfers. The framework captures *data transfers based on adequacy decisions and appropriate safeguards (Standard Contractual Clauses (SCCs) and certification mechanism)*. To the best of our knowledge, this is the first framework that models the GDPR notion of *international data transfers*.
- We apply the framework to an example based on WhatsApp's privacy policies and showcase the capabilities of bigraphs to model complex systems diagrammatically.
- We define the GDPR requirements for international data transfers using CTL and show how they can be proven *formally and automatically* using PRISM.
- We define sorting schemes that ensure the model is well-formed by assigning types to entities and links, effectively preventing nonsensical or undesirable model configurations.

[1] This is before considering the US as an adequate country.

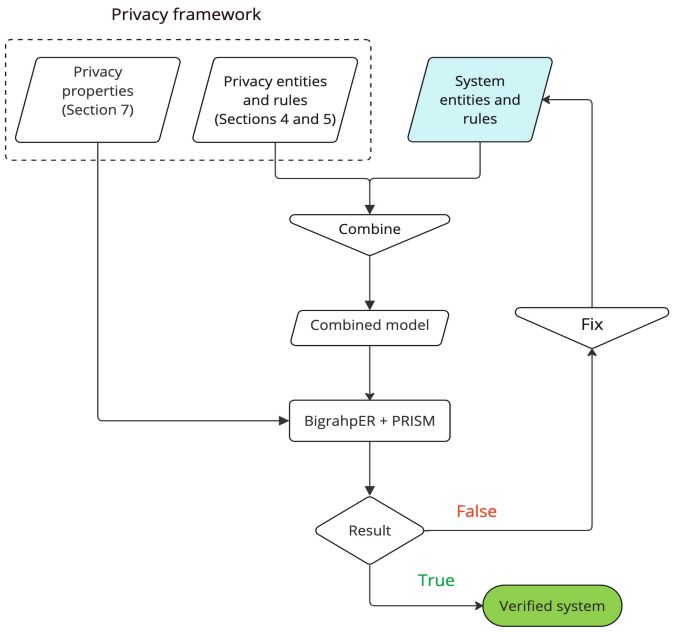


Figure 1: Overview of our bigraph-based privacy framework: The teal box represents the system entities and rules that end-users should specify. The process starts by combining the privacy framework with the system's entities and rules (via links) into a unified model. This combined model is then analysed using BigraphER and the PRISM model checker to prove the privacy properties. If the verification result is true, it indicates that the system meets the GDPR requirements for cross-border data transfers. If false, the system model should be fixed to address the identified issues, after which it is reanalysed. This loop continues until the system passes verification.

The rest of this paper is structured as follows: Section 2 presents the GDPR requirements for international data transfers, while Section 3 gives an overview of Bigraphical Reactive Systems. Section 4 and Section 5 introduce the proposed privacy model, followed by Section 6, which demonstrates how we integrate the privacy framework with the specific system. We discuss the formal verification of the model in Section 7, while Section 8 displays the defined sorts of our model. We introduce an example of detecting privacy violations in Section 9. Section 10 and Section 11 present the related work and the conclusion, respectively.

2 Privacy Regulations for Cross-Border Data Transfers

Many governments have imposed legislation to regulate transferring data outside their countries or territories. They specify certain conditions to transfer data internationally as such transfers could affect their citizens' privacy or national security (especially if the transferred data is sensitive) [10].

For example, the GDPR only allows international transfers in the following three cases:

1. **Adequacy decisions:** this requirement ensures that the recipient country provides a suitable level of data protection. The European Commission has specified a set of countries that provide an adequate level of protection, *e.g.* US, Canada, Japan *etc*. [2].

[2]To view the full list of countries, see `https://t.ly/adequacy-decisions`.

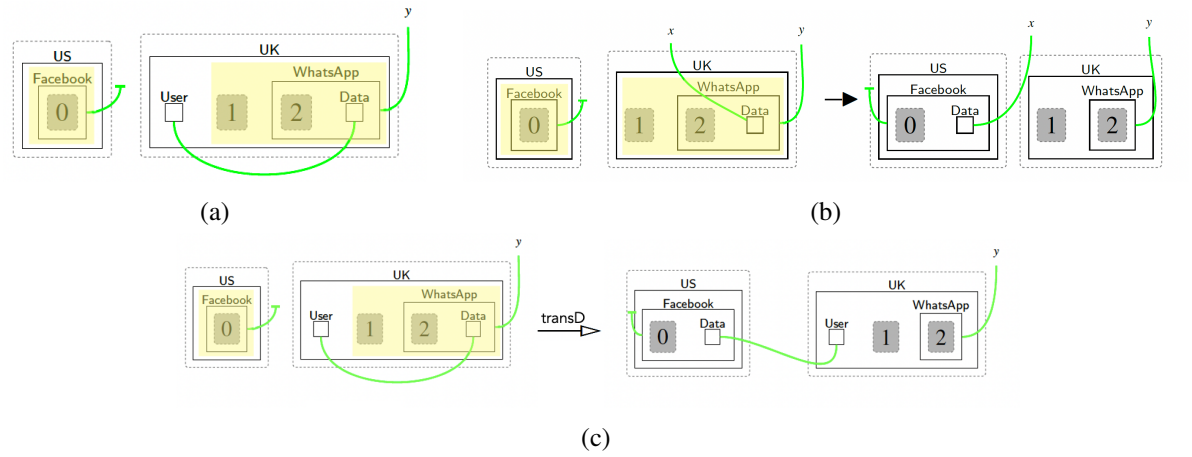


Figure 2: (a) A bigraph representing the initial state of a WhatsApp system that transfers Data to Facebook; (b) Rule `transD` transfers the Data from WhatsApp to Facebook; (c) The result of applying rule `transD` to the initial state, where Facebook acquires the Data.

2. **Appropriate safeguards:** if the adequacy decision does not cover the receiver country, then it should provide appropriate safeguards. The GDPR specifies many safeguards. One is adopting *Standard contractual clauses (SCCs)*, a set of predefined standards and rules identified by the European Commission to protect users' data when it is transferred outside EU countries. These rules should be incorporated into the contract between the sender and receiver organisation [11]. Another safeguard is using the *certification mechanism*. The certification should meet certain criteria (scheme) related to transparent processing and users' rights, *e.g.* clarifying the storage mechanisms of the data, data subjects rights to access *etc*. The certification also should be approved by certification bodies that the GDPR specifies, *e.g.* the European Data Protection Board (EDPB) [8].
3. **Derogations:** the GDPR identifies some special situations in which data transfer is permitted, *e.g.* when transferring data relates to the public interest [3].

This paper models the transfer based on the adequacy decision and the appropriate safeguards. We focus on modelling the *SCCs and certification mechanism* as the SCCs are commonly used among organisations, and the certification should meet some criteria that need to be checked [9].

3 Bigraphical Reactive Systems

An initial bigraph and a set of reaction rules specifying the systems' evolution over time define Bigraphical Reactive Systems (BRSs). Unlike other formalisms, BRSs model systems diagrammatically and algebraically. In this paper, we only use the diagrammatic representation, but the equivalent algebraic specification is available online [2].

An example bigraph is in Fig. 2a: a WhatsApp system that transfers User's Data from the UK to Facebook in the US. Shapes represent entities, *e.g.* US, UK, *etc*. The shaded rectangles are *sites* that represent components that have been abstracted away as they are irrelevant to the context, *e.g.* site 0 could be a data centre of Facebook.

[3] For further details, see `https://www.edpb.europa.eu/sme-data-protection-guide/international-data-transfers_en`.

The dashed rectangles are called *regions*. Each parallel region can be considered as a modelling perspective that is used to split between different concerns [6, 29], *e.g.* the US is modelled in one perspective and the UK in another.

The green *links* are used to connect entities with each other. Links can be open to indicate the possibility of being linked to other unspecified entities *e.g.* link y, or closed to express for example exlusive ownership, *e.g.* the link that connects the User with the Data. Another way to close the links is making them one-to-zero hyperedge, *e.g.* the link that is attached to Facebook. We sometimes use coloured links for readability and visual clarity.

Bigraphs can evolve over time by using reaction rules. From now on, we will use rules to indicate reaction rules. Each rule ($L \longrightarrow\!\!\blacktriangleright R$) contains two parts: the left-hand side (L) and the right-hand side (R). The left-hand side represents the pattern that will be changed, whereas the right-hand side represents the changed pattern. We use the notation $B \xrightarrow{r} B'$ to denote that B rewrites to B' when $r : L \longrightarrow\!\!\blacktriangleright R$ is applied.

For example, rule `transD` in Fig. 2b models a transfer of the user Data to the US. By applying this rule to bigraph B shown in Fig. 2a, the part of the bigraph that matches the left-hand side of the rule (highlighted in yellow) is changed to match the right-hand side [4]. Fig. 2c shows the result of applying rule `transD` (Fig. 2b) to bigraph B (Fig. 2a). The match is performed based on the structure of the bigraphs, not the links' names. This means the links' names are not considered for matching the left-hand side of the rule, *e.g.* changing link y in rule `transD` (Fig. 2b) to z does not affect the application of the rule.

Instantiation maps are a feature of BRS that allows us to copy, swap, or delete sites when the rules are applied. In this paper, we number sites to represent the instantiation maps. For instance, the sites are copied if shown in the left-hand side and the right-hand side of the rule as presented in Fig. 2b. Conversely, they are deleted if they appear in the left-hand side but do not exist in the right-hand side.

We specify and analyse our models using BigraphER [28], an open-source tool for modelling, rewriting, and visualising bigraphs. BigraphER offers two useful features: (1) enforcement of ordering over the rules via priority classes, *i.e.* each class is a set of rules and any rule in a class with lower priority can only be applied when no rules in any higher priority class can be applied, and (2) parameterized entities and rules, *e.g.* Data(x) and `transD`(x), where x can be `Name`, `ID`, or `Age`, to allow sets of entities and rules (each entity or rule using one specific value for x).

4 Modelling Privacy Visually

We present our approach by applying it to an example based on WhatsApp's privacy policies. WhatsApp has two *data controllers* [34], one is located in Ireland to provide services to users in the EU countries, while the other is located in the US to serve users from other countries. These two controllers need to share users' data with the parent company Meta, its data centres, and Facebook's branches worldwide. For space, we present the data centre that is located in Ireland and Facebook's branches in Dubai and Mexico. A partial model of the example is shown in Fig. 3, while the full model is in [2]. The model consists of three perspectives: WhatsSystem, SRType (for sender and receiver), and Locations. The specific system is modelled in the WhatsSystem perspective. It consists of the data controllers in the US and Ireland (Whats), the Facebook branches in Dubai and Mexico (FB), Meta company in the US, and the data centre in Ireland (DC). These entities are linked to their types in SRType perspective, *e.g.*

[4] As this is an illustrative example, we transfer (or optionally duplicate) the data to the US without verifying the GDPR requirements for international data transfers.

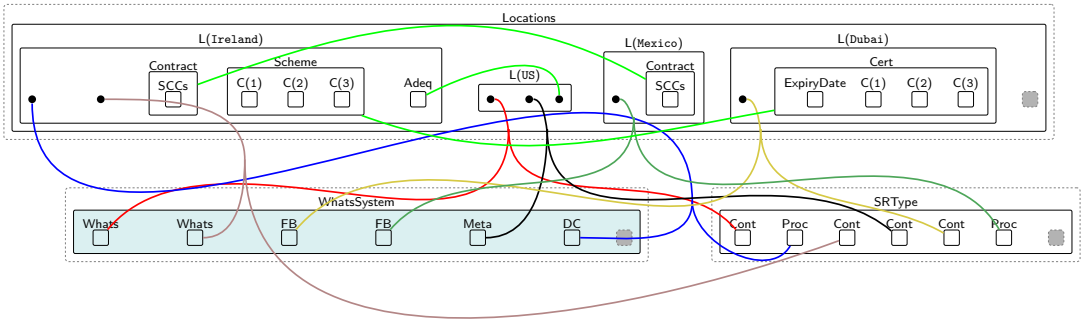


Figure 3: A partial initial state for the WhatsApp example. System-specific entities are shaded in teal. The predefined privacy model appears within the uncoloured regions.

data controller (Cont) or processor (Proc), and to their locations in Locations perspective as discussed in Section 4.1 and Section 4.2.

4.1 Specifying Sender and Receiver Types

SRType perspective is used to specify the types of each system-specific sender and receiver by means of linking, as shown in Fig. 3. This perspective can only contain entities of types *data controller* (Cont) and *processor* (Proc). According to the GDPR, the *data controller* determines the privacy policies and the purposes of processing users' data, whereas the processor processes the data based on the controller's instructions [18]. This perspective can easily be extended to support other types, *e.g.* sub-processor, joint controller *etc*. In Fig. 3, the system's entities that have the controller role (Whats, FB in `Dubai`, and Meta) are linked to their own Cont. Similarly, the system's processors (DC and FB in `Mexico`) are linked to their own Proc.

4.2 Specifying Sender and Receiver Locations

After assigning types to the sender and receiver entities, we need to specify their locations in the Locations perspective. We do so by linking them to an entity of type P (called *pointer*) nested within a location L(x). This is shown in Fig. 3 where, for instance, DC is linked to a P within L(`Ireland`). This approach allows for a great modelling flexibility as the number of entities in each location does not need to be specified and fixed beforehand. Also, by toggling between types P, P′ and Ps, it is easy to define rules to check the adequacy decision and the safeguards when the sender and receiver are in different regions as we will explain in Section 5.1.

Table 1 describes the entities of this perspective. The parameterised entity L(x) allows end-users to add new countries, *e.g.* L(`Japan`). The entity SCCs is nested within Contract. The Contract and Cert are nested within L(x) to indicate that the safeguard provided by the organisation in country x is either the Standard contractual clauses or certification, respectively. Consider the example in Fig. 3. The entity L(`Mexico`) contains SCCs, which is nested within the Contract, meaning that the safeguard provided by Facebook in Mexico is the SCCs. Cert is nested within L(`Dubai`) to indicate that Facebook's branch

Table 1: Locations perspective entities.

Entity	Description
$\mathsf{L}(x)$	Locations as parameterised entities where x is a country's name, *e.g.* `UK`.
Adeq	Linked to adequate countries.
Contract	The contract between the sender and the receiver.
SCCs	The Standard Contractual Clauses.
Scheme	The scheme specified by the GDPR, which the certification should comply with.
ExpiryDate	The expiry date of the certification.
$\mathsf{C}(x)$	Criteria that should be checked to determine the valid certification. Parameter x is a criterion's identifier, *e.g.* `1`, `2`, `Transparency`, `Right-to-access`.
Cert	The certification provided by the receiver.
P, P′, Ps	Pointers for each entities within a location. Shown as solid black, blue and yellow bullets, respectively.

in Dubai uses the certification safeguard. We do not allow the case when a company provides both safeguards as this is not included in the GDPR. Since, `Ireland` is the sender country in our example, it includes Adeq, Contract, and Scheme. The scheme contains three criteria, denoted as $\mathsf{C}(x)$, where $x \in \{1,2,3\}$.

5 Checking privacy requirements

The modelling strategy we have described so far allows to only represent the status of a system at a given time. To model the temporal evolution of a system, we have also to define a set of reaction rules. In this section, we introduce the reaction rules encoding the processes required to check various privacy requirements. Unlike system-specific rules (see Section 6), they can be utilised across systems without changes.

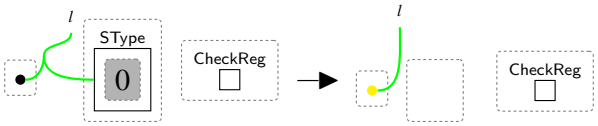
5.1 Checking Regions

To check if data transfers are restricted, we need to check the region of the sender and the receiver. If both are located in the same region, the transfer is considered safe. Otherwise, it is a restricted transfer, and we need to check the adequacy decision and the safeguards.

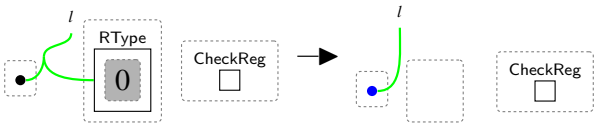
The mechanism for checking the region is to tag the pointers linked to the sender and recipient **type**. As these pointers are nested within the entity $\mathsf{L}(x)$, tagging them enables us to specify if the sender and receiver are in the same or different regions as explained in Section 5.2.

Rule `checkSType` (Fig. 4a) specifies the region of the sender, while rule `checkRType` (Fig. 4b) specifies the receiver's location. The entities SType, RType, and CheckReg are generated via system rule(s) in the system perspective, as outlined in Section 6. SType and RType represent the types of the sender and receiver, respectively (*e.g.*, site 0 can be either a Cont or Proc), and they are linked to the corresponding location of each. The entity CheckReg is generated by a system rule to trigger the privacy model.

Consider the controller Whats, located in `Ireland`, which needs to transfer data to the processor DC, also located in `Ireland`. To model this, we define system rules that specify the type of the sender (Whats) by generating SType around Cont and linking it to the hyper-edge that connects Cont with its



(a) checkSType: tagging the sender's pointer.



(b) checkRType:tagging the receiver's pointer.

Figure 4: The process of checking the sender's and receiver's regions by tagging the pointers linked to their specified types.

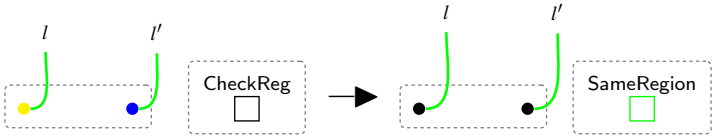


Figure 5: sameReg: the sender and the receiver are in the same region.

pointer in L(Ireland). Similarly, we specify the type for DC, generating RType around Proc, as DC functions as a processor. Additionally, we generate the entity CheckReg to initiate the checking process.

As we generate the SType and RType entities and link them to the locations of the specified types, we apply rule checkSType and checkRType (Fig. 4) to tag the pointers linked to SType and RType, represented by yellow and blue bullets, respectively.

5.2 Entities are in the Same Region

If the tagged pointers (the yellow and blue pointers) are nested within the same region, then the transfer is safe. For instance, the tagged pointers in Section 5.1 are nested within L(Ireland), so we can verify that the sender and the receiver are both in Ireland. The rule that models the result of checking the regions is sameReg (Fig. 5). The entity CheckReg is replaced with SameRegion to indicate the result of checking the region, *i.e.* the sender and the recipient are in the same region, and terminate the checking process. We also untag the pointers to recheck the regions if needed.

5.3 Checking Adequacy

Suppose the data sender is Whats in Ireland, and the receiver is Meta in the US. By checking their regions as explained in Section 5.1, we tag the pointers that are nested within the entities L(Ireland) and L(US), respectively. In such a case, the transfer is restricted because the tagged pointers are nested within different regions, so we must check the adequacy.

Rule checkingAdeq in Fig. 6 checks the adequacy requirement. This rule checks if there is a pointer in the relevant country linked to Adeq. If such a pointer exists, then the country is adequate. The entity Adequate is generated to show the result of checking the adequacy requirement. The sender's and the receiver's pointers (the yellow and blue bullets, respectively) are untagged to allow reuse of the rule if needed. Importantly, failing to find the match of the left-hand side of this rule means the country is

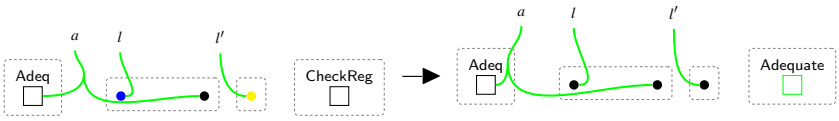


Figure 6: `checkingAdeq`: checking the adequacy of the data importer country.

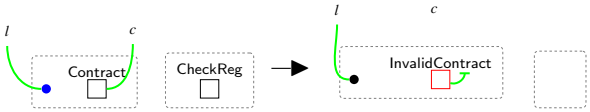


Figure 7: `checkingSCCs`: tagging the contract as invalid if it does not include the SCCs.

inadequate, so we should start checking the safeguards as shown in Section 5.4 and Section 5.5 which have higher priority than rule `checkingAdeq` (Fig. 6).

5.4 Checking SCCs

As mentioned previously, we must check the safeguards when rule `checkingAdeq` (Fig. 6) is not applied. In case the provided safeguard is SCCs, we use rule `checkingSCCs` in Fig. 7 to check whether the receiver country incorporates the SCCs in their Contract.

For example, the Contract of FB in `Mexico` includes SCCs and is linked to the entity Contract nested within L(`Ireland`), as shown in Fig. 3, indicating that `Mexico` agrees to adhere to the SCCs specified by the sender, thereby confirming the validity of the Contract.

Conversely, if the Contract in the receiver country does not include SCCs, rule `checkingSCCs` (Fig. 7) is applied to tag the Contract as InvalidContract. This rule also results in the closure of the link connected to the entity Contract in `Ireland`, *i.e.* link c.

5.5 Checking Certification

If the provided safeguard is a certification (Cert), it must be checked to prove that it meets the Scheme specified by the GDPR. To do so, we use rule `tagCriteria(x)` shown in Fig. 8 to check each criterion (x). Suppose that the Scheme has three criteria: C(1), C(2) and C(3). We replace the parameter x in rule `tagCriteria(x)` (Fig. 8) with the number of criterion, *e.g.* `tagCriteria(1)`, `tagCriteria(2)` *etc*.

Each rule checks the existence of the criterion in the Scheme and the provided Cert, *e.g.* checking that C(1) is nested within the Scheme and the Cert, by tagging the existing criterion (the green entity). Although the GDPR defines more than three criteria, we model only three as the modeller can extend the model by adding x number of criteria, *e.g.* C(4), C(5) $\cdots$ C(x).

If **one** of the defined family rules is not applied, the Cert does not meet the criteria. For example, if rule `tagCriteria(1)` is not applied but rule `tagCriteria(2)` is applied, it means the Cert does not fulfil the first criterion (C(1)). In such a case, we use rule `tagInvalidCert` (Fig. 9) to tag the Cert as invalid (InvalidCert) and close link *s* that is connected to the Scheme. We omit CheckReg to end the checking process.

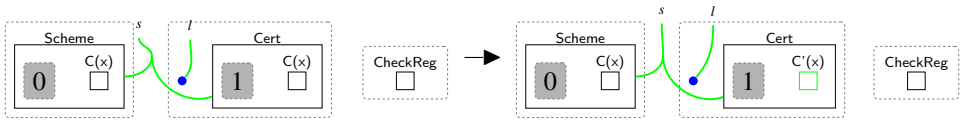


Figure 8: `tagCriteria(x)`: tagging each criterion that must be satisfied.

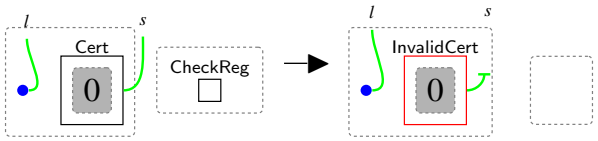


Figure 9: `tagInvalidCert`: tagging the certification as invalid if at least one criterion is not satisfied.

5.6 Result of Checking Certification

Rule `checkingCertResult` (Fig. 10) shows the result of checking the criteria. If the Cert meets the criteria, *i.e.* C(1), C(2) and C(3) are tagged, the Cert is tagged as CompliantCert.

Based on the GDPR, the certification (Cert) is valid for only **three years** [14]. This requires checking its validation date as well [8]. Rule `checkingCertResult` (Fig. 10) initialises the process of checking the expiration by replacing the entity CheckReg with CheckExp and specifying the current date (CurrentDate) to compare it with the expiry date of the Cert as shown in Section 5.7

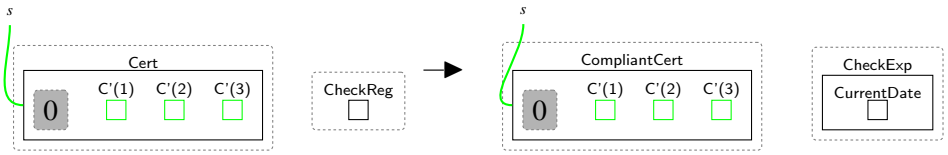


Figure 10: `checkingCertResult`: result of checking the certification and initialising the process of checking its expiry date.

5.7 Checking the Expiry Date

As shown in Fig. 11, we can check the validation date of the CompliantCert by comparing the current date (CurrentDate) with the expiry date (ExpiryDate). If the ExpiryDate is **greater than** (Greater) the CurrentDate, it means the CompliantCert is expired and cannot be used as a safeguard. Otherwise, it is valid and can be used to transfer the data [5].

Rule `ExpiredDate` (Fig. 11) tags the CompliantCert as InvalidCert and removes link *s* connecting the CompliantCert to the Scheme if the ExpiryDate is Greater than the CurrentDate. The entities CheckExp and CurrentDate are omitted to terminate the checking process.

5.8 Withdrawing Certification

Based on the GDPR, the provider of the compliant certification (CompliantCert) has the right to withdraw it [8]. Rule `processWithdReq` (Fig. 12) models the provider's withdrawal request by closing link

[5]To generalise the model, we do not use mathematical computation to perform the comparison. However, we model the two cases: when the certification is expired and when it is valid. See the full model [2].

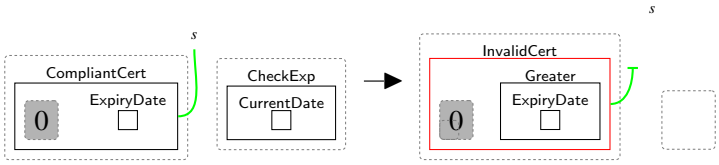


Figure 11: `ExpiredDate`: tagging the expired certification.

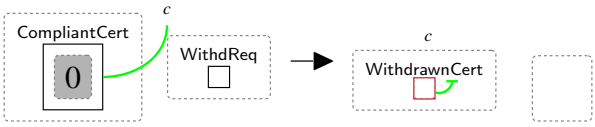


Figure 12: `processWithdReq`: processing the withdrawal request for the certification.

c connecting the CompliantCert to the Scheme and replacing CompliantCert with WithdrawnCert. The entity WithdReq is generated through a system rule (see rule `ReqWithdCert` in Fig. 25).

6 Integration of Privacy Models and Specific System

We introduce an example to show the privacy model's usability and how we can merge it with the specific system. Suppose that the data centre of WhatsApp in `Ireland` (DC) needs to transfer users' data to WhatsApp's data centre in `Singapore` (DC) and to the Facebook branch in `China` (FB). Fig. 13 shows a partial initial state for this example. We consider the DC in `Singapore` to be a processor (linked to Proc) that has a Contract with the DC in `Ireland`. FB is considered as a data controller (linked to Cont), and the safeguard that it provides is a certification (Cert).

We need to trigger the privacy model to check the sender and receiver regions and their safeguards. To do so, we should specify the type of the sender (SType) and the receivers (RType) and generate the entity CheckReg via system rules. Rule `dcIrType` (Fig. 14) specifies the type of DC in `Ireland`. As DC is the sender and is linked to Proc, SType is generated around Proc to indicate that the sender is a processor. We also link SType to the sender's pointer in L(`Ireland`). The entity StartTransfer is generated by a system rule (shown in Fig. 24 Appendix A). It is nested within DC as DC is the entity

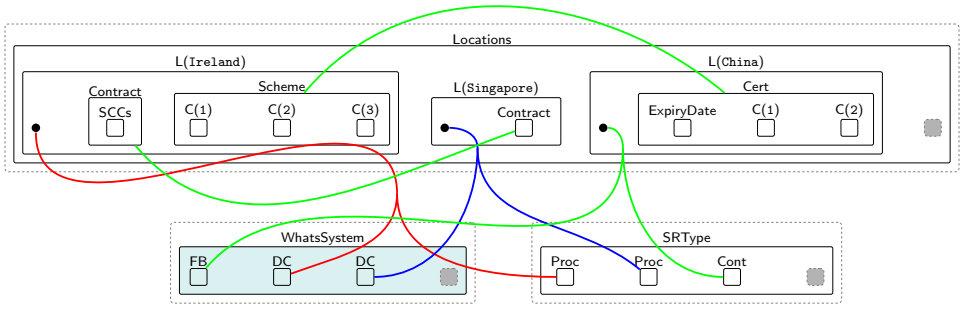


Figure 13: A Partial initial state for the example of transferring data to the data centre in Singapore and to Facebook in China.

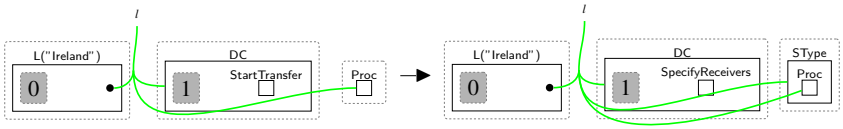


Figure 14: `dcIrType`: specifying the type of the sender (DC in `Ireland`).

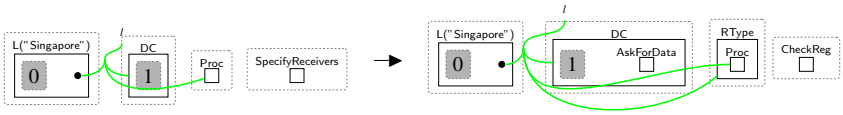


Figure 15: `dcSingType`: specifying the type of the receiver (DC in `Singapore`).

initiating the transfer. StartTransfer is replaced with SpecifyReceivers to specify the receivers types before transferring the data.

Rules `dcSingType` (Fig. 15) and `FacebookCType` (Fig. 16) specify the type of the receivers in `Singapore` (DC) and `China` (FB), respectively. Since DC is linked to Proc, the RType is generated around Proc and linked to DC's location. The entity AskForData represents the receiver's request for the data. FB is a data controller, so RType is generated around the token Cont and linked to `China`. We also replace SpecifyReceivers with CheckReg.

After specifying the sender and receivers types and generating CheckReg, we can start checking the regions by tagging the pointers linked to the specified types. We specify the region of the sender DC by using rule `checkSType` (Fig. 4a). We also use rule `checkRType` (Fig. 4b) to check the region of the receivers DC and FB. By applying these rules, the pointers that will be tagged are the pointers that are located in L(`China`), L(`Singapore`) and L(`Ireland`) as shown in Fig. 17. We can observe that the tagged pointers are located in different countries, so the transfer is restricted.

Based on Fig. 13, Singapore and China are not specified as adequate countries, *i.e.* no pointers in Singapore or China linked to Adeq, as is the case with the `US` in Fig. 3, meaning that the safeguards must be checked. The entity Contract that is nested within L(`Singapore`) does not include SCCs. This indicates that DC in `Singapore` does not incorporate the SCCs in their contract, *e.g.* does not agree to adhere to the SCCs. Rule `checkingSCCs` (Fig. 7) is applied to tag the Contract as InvalidContract and close its link connected to the entity Contract in `Ireland`. The entity InvalidContract is used as a flag to prevent the transfer and block the DC in `Singapore` through the system rule `preventSing` (Fig. 18). Similarly, FB uses the certification mechanism that has only two criteria (C(1) and C(2)) as shown in Fig. 13. By applying rule `tagCriteria(x)` (Fig. 8), only two criteria are tagged. In this case, rule `InvalidCert` (Fig. 9) tags the Cert as InvalidCert. To block FB, we define a system rule that matches explicitly on InvalidCert as shown in Fig. 19. Because we prevented the transfer to DC and FB, we can safely discard the entities InvalidContract and InvalidCert.

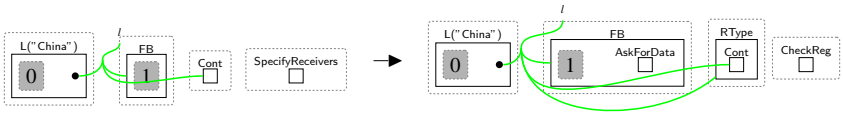


Figure 16: `FacebookCType`: specifying FB type.

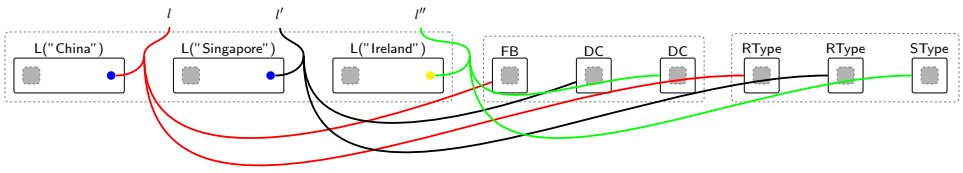


Figure 17: The tagged pointers after specifying DC in `Ireland` and `Singapore` and FB locations.

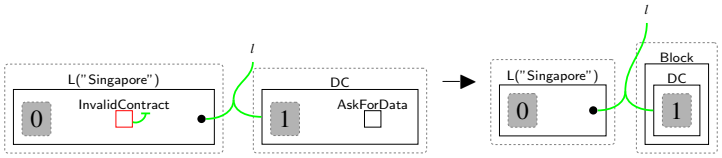


Figure 18: `preventSing`: preventing the transfer to the DC in `Singapore`.

7 Verification

The aim of defining the privacy model is to generate a transition system used to conduct the *verification* step, *i.e.* proving the GDPR requirements for international data transfer within the system. BigraphER utilises the initial state, *e.g.* Fig. 3, and the reaction rules (privacy rules and system rules) to *automatically* generate the transition system. The produced transition system consists of states (bigraphs) that describe the system configurations. Each state is a result of applying a rule (transition). Manually checking privacy properties, *i.e.* the GDPR requirements, within the transition system is challenging as the number of states can be considerable. For example, in the WhatsApp example, the number of the generated states is 193, while the number of the transitions is 267. This complexity would increase if we model a more complex system with additional reaction rules or locations.

Model checkers are tools used to *automatically* prove properties within the transition system. We use the PRISM [22] model checker as BigraphER supports it. To use PRISM, the properties that we aim to check should be encoded using a logic language. We use a non-probabilistic temporal logic language as the requirements we aim to check are specified based on *regulations*. In particular, we use Computation Tree Logic (CTL) [12] to encode the GDPR requirements for international data transfer because CTL quantifies the properties over all paths. This allows us to prove that the GDPR requirements hold across all possible execution paths. We also define a set of predicates using BigraphER to label the states. These predicates represent the right-hand sides of the reaction rules, and PRISM utilises them to parse the transition system. For instance, we define predicate `invalidContra` (Fig. 20) to label the states that

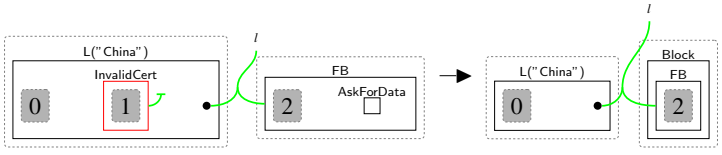


Figure 19: `preventChina`: preventing the transfer to the FB in `China`.

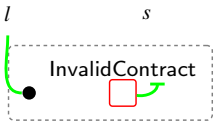


Figure 20: Bigraph pattern of the `invalidContra` predicate.

have the match of the right-hand side of rule `checkingSCCs` (Fig. 7).

The first requirement that we need to check is the adequacy requirement: *no data transfer if the country is inadequate*. The CTL syntax of this requirement is:

$$\mathbf{A}\,[\,\mathbf{G}\,(\,\neg\texttt{adequateCountry} \implies \neg\texttt{dataTransfer})\,] \tag{1}$$

`adequateCountry` is a predicate that labels all states that contain the match of the right-hand side of rule `checkingAdeq` (Fig. 6). The second predicate is `dataTransfer`, which labels the states that transfer the data to the receiver via system rules. **A** is a path quantifier, meaning *for all paths*. **G** is a state quantifier and it means *for all states* (globally). **AG** means the property should hold for all paths globally throughout the entire transition system. Here, we check that: for all paths (**A**), if `adequateCountry` does not hold ($\neg$), `dataTransfer` should not hold ($\neg$) in all states along that path (**G**).

The second requirement that we need to check is: *no data transfer if the SCCs are invalid*:

$$\mathbf{A}\,[\,\mathbf{G}\,(\texttt{invalidContra} \implies \neg\texttt{dataTransfer})\,] \tag{2}$$

As previously mentioned, `invalidContra` is a predicate that holds if the Contract is invalid, so in this case we check: for all paths of the transition system, if `invalidContra` holds in a state, `dataTransfer` should not hold in the same state throughout the path.

The third requirement is: *no transfer should be performed if the certification is invalid*:

$$\mathbf{A}\,[\,\mathbf{G}\,(\texttt{invalidCert} \implies \neg\texttt{dataTransfer})\,] \tag{3}$$

This property proves that: it is always the case that if the certification is invalid, *e.g.* expired or does not meet the criteria, the transfer is never performed.

The last requirement is: *no transfer should be performed if the certification is withdrawn*:

$$\mathbf{A}\,[\,\mathbf{G}\,(\texttt{withdrawCert} \implies (\mathbf{X}\,\neg\,\texttt{dataTransfer}))\,] \tag{4}$$

Here, we prove that for all paths, if `withdrawCert` is true, then `dataTransfer` should be false in all subsequent states on that path. Since `withdrawCert` only becomes true when a system rule generates WithdReq, as discussed in Section 5.8, we apply the next-state operator **X** to ensure that `dataTransfer` is false starting from the state immediately following the state that holds `withdrawCert`. This use of **X** reflects the system's sequential processing order.

8 Sorting

Sorting schemes ensure that bigraphs are well-formed by imposing constraints that prevent undesired or nonsensical models [25], *e.g.* preventing scenarios where Adeq is incorrectly nested within Scheme, or where Cert is linked to FB instead of Scheme[6].

[6]Currently, no automated tool supports these sorting schemes.

Table 2: Description of sort notations.

Notation	Example	Description
s^*	$A\ s^*$	The entity A has *zero* or more children of sort s.
$s \times s^*$	$A\ s \times s^*$	The entity A has at least *one* child of sort s.
$s \times i$	$A\{a \to s \times i\}$	The entity A has a port of sort a, which is connected to ports of sorts s AND i.
$s + i$	$A\{a \to s + i\}$	The entity A has a port of sort a, which is connected to a port of sort s OR i.
$s + 1$	$A\ s + 1$	The entity A has children of sort s OR it does not have any (*i.e.* it is empty).

We use the notation and grammar presented in [4] to define our sorts. Sorts can take two forms: `sort y` or `sort b = A`. The former specifies the sort of link ports, while the latter indicates that A is an atomic entity with sort `b`. If the entity A is linked to other entities and has children, its sort pattern is defined as:

$$\texttt{sort b} = \texttt{A}\{\texttt{y} \to \texttt{z}^*\}\ \texttt{x} + \texttt{w}$$

The sort `y` represents the sort of the port on A that connects to zero or more ports of sort `z`. Meanwhile, `x` and `w` refer to the sorts of A's children, *i.e.* A has children of sort `x` *or* `w`. Table 2 describes through some examples the sort notations used in this paper.

To accurately model privacy and system entities, along with their associated behaviours, we define the necessary sorts for maintaining the integrity of the model and preventing ill-structured configurations across all perspectives.

SRType Perspective. We define the following sorts to ensure a well-formed perspective:

$$\begin{aligned}
&\texttt{sort a},\ \texttt{sort p},\ \texttt{sort et},\ \texttt{sort tp},\ \texttt{sort sy}\\
&\texttt{sort sr} = \texttt{Cont}\{\texttt{a} \to (\texttt{p} + \texttt{tp}) \times \texttt{sy} + \texttt{et}\}\\
&\qquad\qquad \mid \texttt{Proc}\{\texttt{a} \to (\texttt{p} + \texttt{tp}) \times \texttt{sy} + \texttt{et}\}\\
&\texttt{sort ent} = \texttt{SType}\{\texttt{et} \to (\texttt{p} + \texttt{tp}) \times \texttt{a} \times \texttt{sy}\}\ \texttt{sr} \mid \texttt{RType}\{\texttt{et} \to (\texttt{p} + \texttt{tp}) \times \texttt{a} \times \texttt{sy}\}\ \texttt{sr}\\
&\texttt{sort srt} = \texttt{SRType sr} \times \texttt{sr}^*
\end{aligned}$$

The entities Cont and Proc, both having the sort `sr`, each have a port of sort `a`. This port is linked to:

- Either a port of sort `p` (for an untagged pointer) or `tp` (for a tagged pointer) along with a system entity via a port of sort `sy`,
- Or to a port of sort `et` on the entity SType or RType.

The sort `ent` represents the sort of SType and RType. As these entities are used to specify the type of the sender and receiver (whether Cont or Proc), each must include at least one child of sort `sr`. Both SType and RType are linked to three ports: one on a tagged or untagged pointer, one on the entity type, and one on the system entity. The entity SRType, with sort `srt`, acts as the parent of Cont and Proc.

This imposes a structural constraint where SRType contains one or more entities of sort sr, as illustrated in Figures 3 and 13.

By defining and applying these sorts, we ensure that site 0, nested within SType and RType in rule checkSType and checkRType (Fig. 4), either represents the entity Cont or Proc. We also ensure that SType and RType are linked to a pointer, with the other end of the link connected to a port on the system entity.

Locations Perspective. Similar to the SRType perspective, we define the following sorts for the entities of the Locations perspectives:

$$
\begin{aligned}
&\texttt{sort c},\ \texttt{sort i},\ \texttt{sort t},\ \texttt{sort p},\ \texttt{sort st},\texttt{sort d}\\
&\texttt{sort cr} = \texttt{C}(1) \mid \texttt{C}(2) \mid \texttt{C}(3)\\
&\texttt{sort tcr} = \texttt{C}'(1) \mid \texttt{C}'(2) \mid \texttt{C}'(3)\\
&\texttt{sort e} = \texttt{ExpiryDate}\\
&\texttt{sort g} = \texttt{Greater e}\\
&\texttt{sort s} = \texttt{SCCs}\\
&\texttt{sort ad} = \texttt{Adeq}\{\texttt{d} \to \texttt{p} \times \texttt{p}^*\}\\
&\texttt{sort pnt} = \texttt{P}\{\texttt{p} \to (\texttt{sy} \times \texttt{a}) + \texttt{d} + \texttt{et}\} \mid \texttt{P}'\{\texttt{tp} \to (\texttt{sy} \times \texttt{a}) + \texttt{et}\} \mid \texttt{Ps}\{\texttt{tp} \to (\texttt{sy} \times \texttt{a}) + \texttt{et}\}\\
&\texttt{sort scm} = \texttt{Scheme}\{\texttt{sc} \to \texttt{c} \times \texttt{c}^*\}\ \texttt{cr} \times \texttt{cr}^*\\
&\texttt{sort certf} = \texttt{Cert}\{\texttt{c} \to \texttt{sc}\}\ (\texttt{cr}^* + \texttt{tcr}^*)\ \times \texttt{e}\\
&\qquad\qquad \mid \texttt{InvalidCert}\{\texttt{c} \to 1\}\ \texttt{tcr}^*\ \times \texttt{e}\\
&\qquad\qquad \mid \texttt{CompliantCert}\{\texttt{c} \to \texttt{sc}\}\ (\texttt{tcr} \times \texttt{tcr}^*)\ \times \texttt{e}\\
&\texttt{sort ctr} = \texttt{Contract}\{\texttt{t} \to \texttt{t} \times \texttt{t}^*\}\ \texttt{s} + 1\\
&\texttt{sort inctr} = \texttt{InvalidContract}\{\texttt{i} \to 1\}\\
&\texttt{sort l} = \texttt{L}\ (\texttt{Ireland})\ (\texttt{pnt}^* \times \texttt{pnt}) \times\ (\texttt{ad} + \texttt{sc} + \texttt{ctr})\\
&\qquad \mid \texttt{L}\ (\texttt{US})\ (\texttt{pnt}^* \times \texttt{pnt})\\
&\qquad \mid \texttt{L}\ (\texttt{Mexico})\ (\texttt{pnt}^* \times \texttt{pnt})\ \times\ (\texttt{ctr} + \texttt{inctr})\\
&\qquad \mid \texttt{L}\ (\texttt{Singapore})\ (\texttt{pnt}^* \times \texttt{pnt})\ \times\ (\texttt{ctr} + \texttt{inctr})\\
&\qquad \mid \texttt{L}\ (\texttt{Dubai})\ (\texttt{pnt}^* \times \texttt{pnt})\ \times\ \texttt{certf}\\
&\qquad \mid \texttt{L}\ (\texttt{China})\ (\texttt{pnt}^* \times \texttt{pnt})\ \times\ \texttt{certf}\\
&\texttt{sort loc} = \texttt{Locations}\ \texttt{l}^* \times \texttt{l}
\end{aligned}
$$

We assign the sort pnt to the pointers P, P′, and Ps, which represent the solid black, blue, and yellow bullets, respectively. Pointer P has a port of sort p, while pointers P′ and Ps have ports of sort tp. These pointers are linked to two ports: one on a system entity (with sort sy) and one on a corresponding entity type within the SRType perspective via a port of sort a. Additionally, pointer P can be linked to a port of sort d on the entity Adeq, or to SType or RType via a port of sort et, while pointers P′ and Ps should not be linked to a port of sort d.

The entity Scheme is linked to one or more certifications (Cert) and must always include at least one criterion of sort cr or tcr. The certifications, whether invalid or compliant, are assigned the sort certf. Both Cert and CompliantCert should be linked to a port on the entity Scheme, and both have a child of sort e. However, Cert may contain zero or more *untagged* or *tagged* criteria, while CompliantCert must always contain at least one *tagged* criterion to satisfy the criteria specified by the Scheme. In contrast, InvalidCert can include zero or more tagged criteria. As shown in Fig. 9 and Fig. 11, InvalidCert has a closed link, denoted by 1 in the defined sorts, indicating the absence of a connection to Scheme.

The sort of Contract is ctr. Since Contract is linked to one or more contracts, the output and input ports share the same sort (t). Contract may either have a child of sort s, or it may be empty, in which

case it can be tagged as InvalidContract by rule checkingSCCs (Fig. 7), which also closes its link.

The parameterised entity L is defined with the sort l. The sorts of its children vary based on its parameters. For example, L(Ireland) is the sender country and must contain a child of sort ad, sc, or ctr to allow the GDPR compliance check.

Receiver countries that use SSCs, *e.g.* L(Mexico) and L(Singapore), must include the entity Contract (sort ctr) or InvalidContract (sort inctr) if the contract is empty, *i.e.* it does not contain SCCs. In contrast, countries like L(Dubai) and L(China) use a certification mechanism, so their children must be of sort certf. If the receiver country is adequate, it does not contain any safeguards, meaning it includes only tagged or untagged pointers.

Regardless of the value of the parameter for entity L, it must always contain at least one pointer (tagged or untagged) that links to system entities and their types. The parent of L is the entity Locations that has sort loc and it should contain at least one child of sort l.

WhatsSystem Perspective. The system consists of various entities such as DC, FB, and others, each of which has specific roles and children entities based on their functions in the data exchange process:

$$
\begin{aligned}
&\texttt{sort snd} = \texttt{SpecifyReceivers} \mid \texttt{CheckReg} \mid \texttt{SameRegion} \mid \texttt{Adequate}\\
&\texttt{sort req} = \texttt{AskForData} \mid \texttt{WithdReq}\\
&\texttt{sort info} = \texttt{CopyInfo}\\
&\texttt{sort cd} = \texttt{CurrentDate}\\
&\texttt{sort chex} = \texttt{CheckExp cd}\\
&\texttt{sort sys} = \texttt{DC}\{\texttt{sy} \to (\texttt{p}+\texttt{tp}) \times \texttt{a}+\texttt{et}\}\ \texttt{snd}+\texttt{req}+\texttt{info}+\texttt{chex}+1\\
&\qquad\qquad \mid \texttt{FB}\{\texttt{sy} \to (\texttt{p}+\texttt{tp}) \times \texttt{a}+\texttt{et}\}\ \texttt{req}+\texttt{info}+1\\
&\texttt{sort b} = \texttt{Block sys}
\end{aligned}
$$

DC and FB are connected by a hyper-edge to a tagged or untagged pointer, as well as to their corresponding types, *e.g.* Cont. When the entities SType and RType are generated, they can be linked to DC or FB via a port of sort et. In our model, DC can act as both a sender and a receiver. When it is a sender, it contains the following entities: SpecifyReceivers, CheckReg or SameRegion. When DC is a receiver, it can contain AskForData or WithdReq. FB and Other companies in the model, *e.g.* Whats, Meta *etc*., contains children such as AskForData or CopyInfo, which are specifically related to their role in requesting and receiving data. The entity Block is used to block companies from accessing the data, thereby always containing one child of sort sys.

By applying these sorts to any initial state modelling the starting configuration of a system and all the reaction rules, we can formally verify the same properties proven by PRISM as discussed in Section 7. For example, we ensure that data transfer occurs only if the Contract is valid, specifically when it includes SCCs. Without the use of these sorts, we risk the Contract containing unintended entities instead of SCCs, which would mean it fails to be marked as InvalidContract and could wrongly allow the transfer to proceed.

9 Privacy Violations Detection

Defining system rules can inadvertently lead to privacy violations. As discussed in Section 6, data must not be transferred to FB in China because the provided certification is invalid. However, modellers may define a system rule that allows data to be transferred to FB, as shown in Fig. 21. This violation occurs because the rule is defined without explicitly matching on InvalidCert. The PRISM model checker detects this violation as Property (3) in Section 7 is not met (*i.e.* the certification is invalid, but the transfer

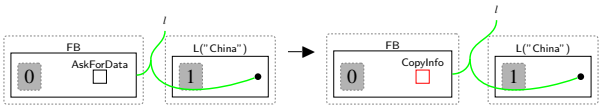


Figure 21: `privacyViolation`: system rule that leads to a privacy violation by transferring the data to FB in `China` .

occurs regardless). By inspecting the generated transition system, it is straightforward to identify the rule introducing the violation, *i.e.* the rule generating a state in which simultaneously `invalidCert` and `dataTransfer` hold. To correct this rule, we redefine it as presented in Fig. 19.

Malformed initial states or reaction rules can introduce privacy violations. For instance, according to the defined sorting constraints, L(`Singapore`) should contain only a single child of either sort `ctr` *or* `inctr`. However, if a malformed initial state or rule assigns L(`Singapore`) both sorts, this could lead to data being transferred to L(`Singapore`) during the execution of the transition system, violating the intended semantics.

Another example that highlights the vital role of sorts in ensuring the correctness of the model is when Cont is linked to a port of sort `sc` instead of `p`. In this scenario, the process of checking the region cannot be performed, resulting in a bug in the model. Such an issue can lead to significant consequences, especially when data transfer is critical. To maintain the integrity of the model, Cont must be linked to ports of sort `p` and `sy`, as specified in the sorts.

10 Related Work

There is a lack of using formal methods to prove systems' compliance with the GDPR requirements for international data transfer. However, multiple works have been proposed to prove the systems' compliance with other GDPR notions. For example, Karami *et al.* [20] define data protection language (DPL) as an object-oriented language that handles the GDPR notions of purpose and storage limitation, the right to be forgotten, and providing/withdrawing consent. They use multiset rewrite rules to formally model the operational semantics of DPL. The resulting model is checked using the model checker Maude [23], but the checking process was *partially automated* as they also provide a pen-and-paper proof. Another tool that checks these notions is the Model-Based approach to Identify Privacy Violations in software requirements (MBIPV) [36]. It is a *fully automatic* tool that detects GDPR violations by converting UML class diagrams into a Kripke model. The NuSMV [1] model checker is used to detect the violations in the defined Kripke model.

Kammueller [19] uses the theorem prover Isabelle (HOL) to prove the compliance of IoT healthcare systems with the GDPR requirements for users' right to access their data, their right to delete their data and the compliance of system functions with these rights. However, the proving process is not entirely automated.

Milner's π-calculus [24] is extended [21] by defining a set of privacy calculus syntax. Later, the privacy calculus is improved to model the GDPR notion of providing and withdrawing consent [32]. Another work based on π-calculus involves using multiparty session types to develop D´alogoP tool that proves the GDPR notion of processing purposes within systems [31].

All the mentioned approaches use formal techniques to evaluate the systems' compliance with specific aspects of the GDPR. However, they are limited in verifying the GDPR requirements for international data transfers because they need to support *spatial properties* as bigraphs. [25].

Recent *informal* approaches are proposed to prove the GDPR requirements for international transfer. Guamán *et al.* [16] define a method that systematically analyses the compliance of Android mobile apps with the GDPR notion of international data transfer. Later, the method is converted to an automated tool [17] but does not handle the certification mechanisms as our model does. Pascual *et al.* [26] provide *Hunter*, an automated tool for tracking anycast communications protocol that routes the data to the nearest server regardless of location. The proposed approach mainly relies on the adequacy decision but does not cover other requirements for international data transfer, *e.g.* the use of safeguards. Unlike our approach, which *formally* proves the GDPR requirements for cross-border data transfers, these methods use *informal* techniques, potentially resulting in a less rigorous analysis of these requirements.

11 Conclusion

We provide a bigraphical framework that captures the GDPR requirements for cross-border data transfer. We encode these requirements using CTL and prove them by PRISM. We also define sorting schemes to ensure that our framework is well-formed.

Although we use only one example to define the framework, we believe it can capture different systems. The multi-perspectives approach enables us to model the GDPR requirements for international data transfers independently from specific systems. This supports the framework's ability to model several systems. Using parameters also allows various developers to model different countries and criteria.

Using our framework requires collaboration between software engineers with a background in bigraphs theory and privacy experts. The engineers build the model and explain the data flow to the privacy experts. The privacy experts advise on the certification criteria and the validity of the SCC, or they even amend the model if the regulations are updated. Organisations can use our framework to prove their adherence to the international transfer requirements.

In future, we aim to extend the framework to capture other safeguards, *e.g.* Binding Corporate Rules (BCRs) and Codes of conduct. We also aim to model the case of transferring data based on the user's consent. The model need to be applied to more examples and real-world systems to show its generality. An automated tool is also needed to define and check sorting schemes automatically.

Acknowledgements

This work is partially supported by an Amazon Research Award on Automated Reasoning.

References

[1] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia & Marco Roveri (2000): *NUSMV: A New Symbolic Model Checker*. International Journal on Software Tools for Technology Transfer 2(4), pp. 410–425, doi:10.1007/s100090050046.

[2] Ebtihal Althubiti & Michele Sevegnani (2024): *Modelling Privacy Compliance in Cross-border Data Transfers with Bigraphs (Latest Version)*, doi:10.5281/zenodo.14052899.

[3] Blair Archibald, Muffy Calder & Michele Sevegnani (2020): *Conditional bigraphs*. In: International Conference on Graph Transformation, Springer, pp. 3–19, doi:10.1007/978-3-030-51372-6_1.

[4] Blair Archibald & Michele Sevegnani (2024): *A Bigraphs Paper of Sorts*. In: Graph Transformation, Springer Nature Switzerland, Cham, pp. 21–38, doi:10.1007/978-3-031-64285-2_2.

[5] Office of the Australian Information Commissioner (OAIC) (2019): *Sending personal information overseas*. Available at `https://www.t.ly/oIcI7`.

[6] Steve Benford, Muffy Calder, Tom Rodden & Michele Sevegnani (2016): *On Lions, Impala, and Bigraphs: Modelling Interactions in Physical/Virtual Spaces*. *ACM Trans. Comput.-Hum. Interact.* 23(2), pp. 9:1–9:56, doi:10.1145/2882784.

[7] European Data Protection Board (2023): *1.2 billion euro fine for Facebook as a result of EDPB binding decision*. Available at `https://www.t.ly/HRqTR`.

[8] European Data Protection Board (2023): *Guidelines 07/2022 on certification as a tool for transfers*. Available at `https://www.t.ly/OPhXd`.

[9] European Data Protection Board (n.d.): *International data transfers*. Available at `https://www.t.ly/HDXOO`.

[10] Francesca Casalini & Javier López González (2019): *Trade and cross-border data flows*. *OECD Trade Policy Papers*, doi:10.1787/b2023a47-en.

[11] Francesca Casalini, Javier López González & Taku Nemoto (2021): *Mapping commonalities in regulatory approaches to cross-border data transfers*. *OECD Trade Policy Papers*, doi:10.1787/ca9f974e-en.

[12] Edmund M Clarke & E Allen Emerson (1981): *Design and synthesis of synchronization skeletons using branching time temporal logic*. In: *Workshop on logic of programs*, Springer, pp. 52–71, doi:10.1007/BFb0025774.

[13] European Commission (n.d.): *Standard Contractual Clauses (SCC)*. Available at `https://www.t.ly/7HnF6`.

[14] Intersoft Consulting (n.d.): *Certification*. Available at `https://gdpr-info.eu/art-42-gdpr/`.

[15] Intersoft Consulting (n.d.): *Transfers of personal data to third countries or international organisations*. Available at `https://gdpr-info.eu/chapter-5/`.

[16] Danny S Guamán, Jose M Del Alamo & Julio C Caiza (2021): *GDPR compliance assessment for cross-border personal data transfers in android apps*. *IEEE Access* 9, pp. 15961–15982, doi:10.1109/ACCESS.2021.3053130.

[17] Danny S Guamán, David Rodriguez, Jose M del Alamo & Jose Such (2023): *Automated GDPR compliance assessment for cross-border personal data transfers in android applications*. *Computers & Security* 130, p. 103262, doi:10.1016/j.cose.2023.103262.

[18] Information Commissioner's Office (ICO) (2023): *How do you determine whether you are a controller or processor?* Available at `t.ly/MPvgu`.

[19] Florian Kammueller (2018): *Formal modeling and analysis of data protection for gdpr compliance of iot healthcare systems*. In: *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, IEEE, pp. 3319–3324, doi:10.1109/SMC.2018.00562.

[20] Farzane Karami, David Basin & Einar Broch Johnsen (2022): *DPL: A Language for GDPR Enforcement*. In: *2022 IEEE 35th Computer Security Foundations Symposium (CSF)*, IEEE, pp. 112–129, doi:10.1109/CSF54842.2022.9919687.

[21] Dimitrios Kouzapas & Anna Philippou (2017): *Privacy by typing in the π-calculus*. *Log. Methods Comput. Sci.* 13(4), doi:10.23638/LMCS-13(4:27)2017.

[22] M. Kwiatkowska, G. Norman & D. Parker (2011): *PRISM 4.0: Verification of Probabilistic Real-time Systems*. In G. Gopalakrishnan & S. Qadeer, editors: *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, *LNCS* 6806, Springer, pp. 585–591, doi:10.1007/978-3-642-22110-1_47.

[23] José Meseguer (2012): *Twenty years of rewriting logic*. *J. Log. Algebr. Program.* 81(7-8), pp. 721–781, doi:10.1016/j.jlap.2012.06.003.

[24] Robin Milner (1999): *Communicating and mobile systems: the pi calculus*. Cambridge university press.

[25] Robin Milner (2009): *The space and motion of communicating agents*. Cambridge University Press, doi:10.1017/CBO9780511626661.

[26] Hugo Pascual, Jose M Del Alamo, David Rodriguez & Juan C Dueñas (2024): *Hunter: Tracing anycast communications to uncover cross-border personal data transfers*. Computers & Security 141, p. 103823, doi:10.1016/j.cose.2024.103823.

[27] Abbas Razaghpanah, Rishab Nithyanand, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Mark Allman, Christian Kreibich & Phillipa Gill (2018): *Apps, trackers, privacy, and regulators: A global study of the mobile tracking ecosystem*. In: The 25th Annual Network and Distributed System Security Symposium (NDSS 2018), doi:10.14722/ndss.2018.23353.

[28] Michele Sevegnani & Muffy Calder (2016): *BigraphER: rewriting and analysis engine for bigraphs*. In: International Conference on Computer Aided Verification, Springer, pp. 494–501, doi:10.1007/978-3-319-41540-6_27.

[29] Michele Sevegnani, Milan Kabác, Muffy Calder & Julie A. McCann (2018): *Modelling and Verification of Large-Scale Sensor Network Infrastructures*. In: 23rd International Conference on Engineering of Complex Computer Systems, ICECCS 2018, Melbourne, Australia, December 12-14, 2018, pp. 71–81, doi:10.1109/ICECCS2018.2018.00016.

[30] European Data Protection Supervisor (n.d.): *International transfers*. Available at `https://www.t.ly/GPuii`.

[31] Evangelia Vanezi, Georgia M Kapitsaki, Dimitrios Kouzapas, Anna Philippou & George A Papadopoulos (2020): *DiálogoP-A Language and a Graphical Tool for Formally Defining GDPR Purposes*. In: International Conference on Research Challenges in Information Science, Springer, pp. 569–575, doi:10.1007/978-3-030-50316-1_40.

[32] Evangelia Vanezi, Dimitrios Kouzapas, Georgia M Kapitsaki & Anna Philippou (2019): *Towards GDPR Compliant Software Design: A Formal Framework for Analyzing System Models*. In: International Conference on Evaluation of Novel Approaches to Software Engineering, Springer, pp. 135–162, doi:10.1007/978-3-030-40223-5_7.

[33] WhatsApp (2024): *WhatsApp Privacy Policy*. Available at `https://www.t.ly/uGjmN`.

[34] WhatsApp (2024): *Who is providing your WhatsApp services*. Available at `https://www.t.ly/NqTfZ`.

[35] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui & John Fitzgerald (2009): *Formal methods: Practice and experience*. ACM computing surveys (CSUR) 41(4), pp. 1–36, doi:10.1145/1592434.1592436.

[36] Tong Ye, Yi Zhuang & Gongzhe Qiao (2023): *MBIPV: a model-based approach for identifying privacy violations from software requirements*. Software and Systems Modeling 22(4), pp. 1251–1280, doi:10.1007/s10270-022-01072-3.

A System Rules

We present some specific system rules for the WhatsApp example:

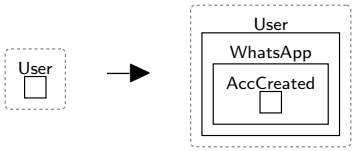


Figure 22: `creatingAccount`: the system's user starts using WhatsApp to create an account.

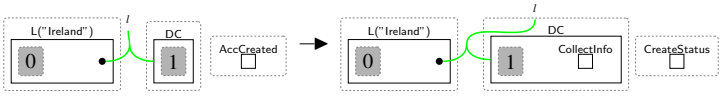


Figure 23: `creatingStatus`: the user creates a status, so the entity AccCreated is replaced with CreateStatus. We assume that DC collects information about the user's status, *e.g.* status content type, location information, information about the device used to post the status *etc*. We model that by generating the entity CollectInfo. We do not specify exactly what the collected data is, as our model focuses on checking the adequacy decision and the safeguards regardless of the type of transferred data.

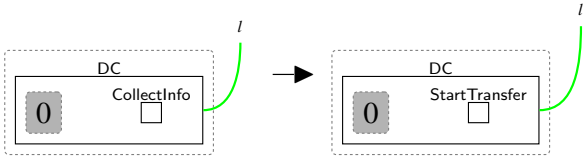


Figure 24: `initialisingTrnas`: DC initialises the transferring process by replacing the entity CollectInfo with StartTransfer to start applying rule `dcIrType` in Fig. 14

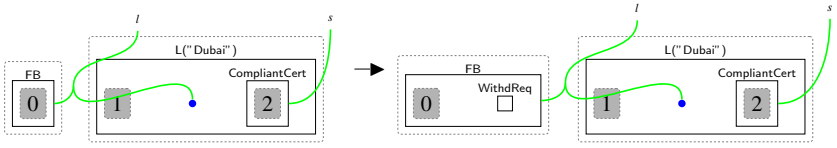


Figure 25: `ReqWithdCert`: FB in `Dubai` asks to withdraw its certification. The rule explicitly matches on the CompliantCert as we need to ensure that the entity requesting the withdrawal is using the certification, not the SCCs.

Linear-Time Graph Programs without Preconditions

Ziad Ismaili Alaoui *
Department of Computer Science, University of Liverpool
Liverpool, United Kingdom
ziad.ismaili-alaoui@liverpool.ac.uk

Detlef Plump
Department of Computer Science, University of York
York, United Kingdom
detlef.plump@york.ac.uk

We report on a recent breakthrough in rule-based graph programming, which allows us to reach the time complexity of imperative linear-time algorithms. In general, achieving the complexity of graph algorithms in conventional languages using graph transformation rules is challenging due to the cost of graph matching. Previous work demonstrated that with *rooted* rules, certain algorithms can be executed in linear time using the graph programming language GP 2. However, for non-destructive algorithms that retain the structure of input graphs, achieving linear runtime required input graphs to be connected and of bounded node degree. In this paper, we overcome these preconditions by enhancing the graph data structure generated by the GP 2 compiler and exploiting the new structure in programs. We present three case studies, a cycle detection program, a program for numbering the connected components of a graph, and a breadth-first search program. Each of these programs runs in linear time on both connected and disconnected input graphs with arbitrary node degrees. We give empirical evidence for the linear time complexity by using timings for various classes of input graphs.

1 Introduction

Designing and implementing languages for rule-based graph rewriting, such as GReAT [1], GROOVE [10], GrGen.Net [11], Henshin [15], and PORGY [9], poses significant performance challenges. Typically, programs written in these languages do not achieve the same runtime efficiency as those written in conventional imperative languages such as C or Java. The primary obstacle is the cost of graph matching, where matching the left-hand graph L of a rule within a host graph G generally requires time $|G|^{|L|}$, with $|X|$ denoting the size of graph X. (Since L is fixed, this is a polynomial.) As a consequence, standard imperative graph algorithms running in linear time (see, for example, [7, 14]) may exhibit non-linear, polynomial runtimes when recast as rule-based graph programs.

To address this issue, the graph programming language GP 2 [13] supports *rooted* graph transformation rules, initially proposed by Dörr [8]. This approach involves designating certain nodes as *roots* and matching them with roots in the host graphs. Consequently, only the neighbourhoods of host graph roots need to be searched for matches, which can often be done in constant time under mild conditions. The GP 2 compiler [3] maintains a list of pointers to roots in the host graph, facilitating constant-time access to roots if their number remains bounded throughout the program's execution. In [4], *fast* rules were identified as a class of rooted rules that can be applied in constant time, provided host graphs contain a bounded number of roots and have a bounded node degree.

The first linear-time graph problem implemented by a GP 2 program was 2-colouring. In [4, 3], it is shown that this program colours connected graphs of bounded node degree in linear time. Since then, the GP 2 compiler has received some major improvements, particularly related to the runtime graph data

*This author's work was done while he was affiliated with the University of York.

J. Endrullis, D. Grzelak, T. Heindel and J. Kosiol (Eds):
GCM 2023 and GCM 2024
EPTCS 417, 2025, pp. 39–54, doi:10.4204/EPTCS.417.3

structure used by the compiled programs [6]. These improvements made a linear-time worst-case performance possible for a wider class of programs, in some cases even on input graph classes of unbounded degree. See [5] for an overview.

Despite this progress, programs that retain the structure of input graphs, such as the aforementioned 2-colouring program, have until now required non-linear runtimes on disconnected graphs. The problem is that, after a connected component is visited, the number of failed attempts to match a non-visited node in a different connected component may increase. Consequently, in disconnected graph classes, this number may grow quadratically in the graph size, leading to a quadratic program runtime. In connected graph classes, this undesirable behaviour is ruled out because all nodes are reachable from a single undirected depth-first search.

In this paper, we present two updates to the GP 2 compiler, one being introduced in [2], which allow lifting the preconditions that host graphs must be connected and have a bounded node degree. In short, the solution is to improve the graph data structure generated by the compiler. Nodes are now stored in separate linked lists based on their *marks* (red, green, blue, grey or unmarked), and each node comes with a two-dimensional array of linked lists storing all incident edges based on their marks (red, green, blue, dashed or unmarked) and orientation (incoming, outgoing or looping). This enables the matching algorithm to find in constant time a node with a specific mark or an edge with a specific mark and orientation. For instance, if a red node is needed, a single access to the list of red nodes will either locate such a node or confirm its absence. Similarly, if an outgoing green edge is required, a single access to the corresponding linked list will either find such an edge or determine that none exists.

In addition to the new graph data structure, a programming technique is needed to take advantage of the improved storage. In a case study, we demonstrate how to recognize acyclic graphs in linear time with the new graph representation. Our program admits both connected and disconnected input graphs with arbitrary node degrees. It either detects that a graph is cyclic or returns an acyclic graph that is isomorphic to the input graph up to marking. Then, we discuss two more programs relying on the improved compiler: one numbering the connected components of a graph, and another performing a breadth-first search. For both programs, we give empirical evidence that they run in linear time on different classes of input graphs.

2 The Problem with Graph Classes of Unbounded Degree

The current section and the next explain the problem caused by classes of input graphs with an unbounded node degree, and how the updated compiler overcomes this problem. We include this material from [2] for the benefit of the reader. For a description of the GP 2 programming language, we refer to [5].

Previously, non-destructive GP 2 programs based on depth-first-search ran in linear time on graph classes of bounded node degree but in non-linear time on graph classes of unbounded degree [5]. For example, consider the program `is-connected` in Figure 1 which checks whether a graph is connected.[1] Input graphs are arbitrary GP 2 host graphs with grey nodes and unmarked edges. The program fails on a graph if and only if the graph is disconnected.

Rule `init` picks an arbitrary grey node as a root (if the input graph is non-empty) and then the loop `DFS!` performs a depth-first search of the connected component of the node chosen by `init`. The rule `forward` marks each newly visited node blue, and `back` unmarks it once it is processed. Procedure `DFS`

[1] Node labels such as `x` are written inside nodes, whereas small integers below nodes are their identifiers. Nodes without identifiers on the left-hand side are to be deleted; nodes without identifiers on the right-hand side are to be added. Nodes with the same identifier on each side are to be kept.

ends when `back` fails to match, indicating that the search is complete. Rule `match` checks whether a grey-marked node still exists in the graph following the execution of `DFS!`. This is the case if and only if the input graph contains more than one connected component. In this situation the program invokes the command `fail`, otherwise it terminates by returning the graph resulting from the depth-first search.

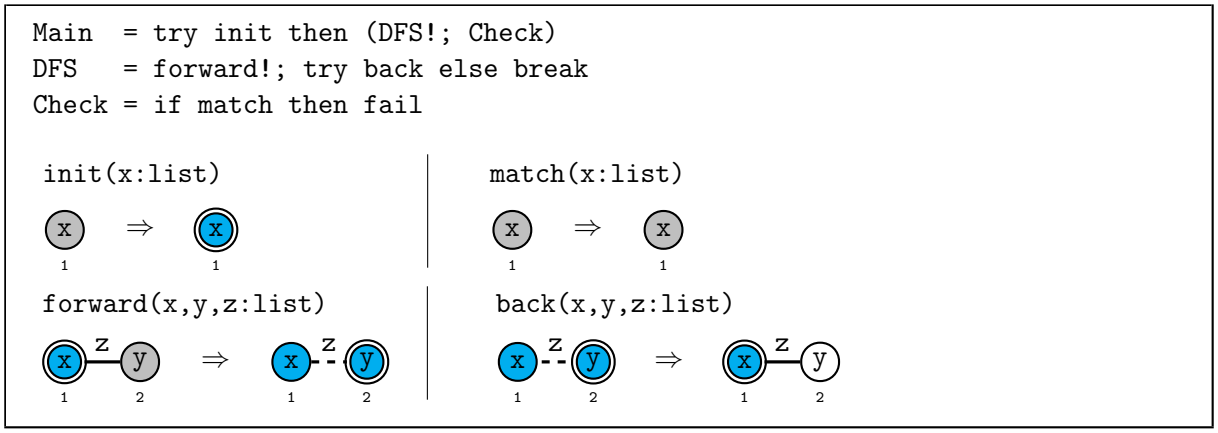


Figure 1: The old program `is-connected`.

It can be shown that the program `is-connected` runs in linear time on classes of graphs with bounded node degree [5]. However, as the following example shows, the program may require non-linear time on unbounded-degree graph classes. Figure 2 shows an execution of `is-connected` on a star graph with 8 edges (see also Figure 13). The numbers below the graphs show the ranges of attempts that the matching algorithm may perform. For instance, in the second graph of the top row, either a match is found immediately among the edges that connect the central node with the grey nodes, or the dashed edge is unsuccessfully tried first. In order to find a match for the rule `forward`, the matching algorithm considers, in the worst case, every edge incident with the root. When the node central to the graph is rooted and the rule `forward` is called, the matching algorithm may first attempt a match with the dashed back edge and all edges incident with an unmarked node. Therefore, the maximum number of matching attempts for `forward` grows as the root moves back to the central node. As can be seen from this example, the worst-case complexity of matching `forward` throughout the program's execution is $2|E| + \sum_{i=1}^{|E|} i = \mathrm{O}(|E|^2)$ where E is the set of edges.

3 First Compiler Enhancement

To address the problem described in Section 2, we changed the GP 2 compiler described in [6], which we refer to as the *2020 compiler*. We call the version introduced in this paper the *new compiler*[2]. The 2020 compiler stored the host graph's structure as one linked list containing every node in the graph, with each node storing two linked lists of edges: one for incoming edges and one for outgoing edges. When iterating through edge lists to find a particular match for a rule edge, the 2020 compiler had to traverse through edges with marks incompatible with that of the rule edge. This resulted in performance issues, especially if nodes could be incident to an unbounded number of edges with marks incompatible with the edge to be matched.

[2]Available at: `https://github.com/UoYCS-plasma/GP2`.

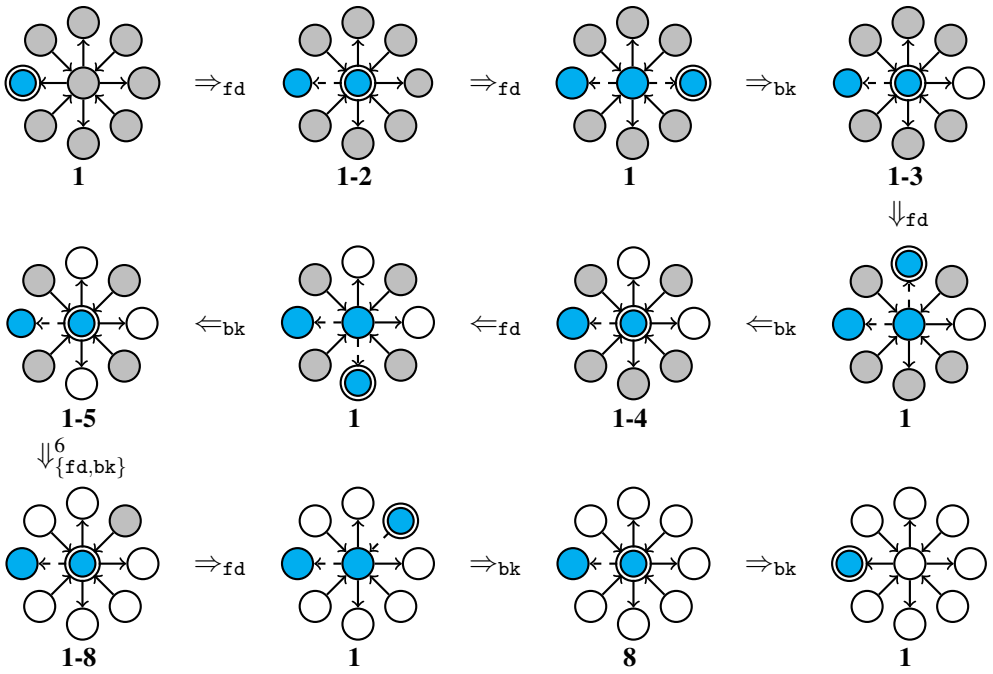


Figure 2: Matching attempts with the `forward` rule. `fd` and `bk` denote `forward` and `back`, respectively.

For example, consider the rule `move` from Figure 8. Initially, the matching algorithm matches node `1` from the interface with a root node in the host graph. Subsequently, it iterates through the node's edge lists to locate a match for the red edge. In the 2020 compiler, all edges incident to this node were stored within two lists, one for each orientation, irrespective of their marks. However, if the node is incident to a growing number of unmatchable edges (because of mark changes), the matching algorithm would face, in the worst case, a growing number of iterations through the edge lists to find a single red edge.

When considering a match for a rule edge, host edges with incorrect orientation or incompatible marks do not match; thus, the matching algorithm need not iterate through them. By organising the edges incident to a node into linked lists based on their mark and orientation, the matching algorithm can selectively consider linked lists of edges of correct mark and orientation. More precisely, in the new compiler, we updated the graph structure of the 2020 compiler by replacing the two linked lists with a two-dimensional array of linked lists of edges. Each element of the array stores a linked list containing edges of a particular mark and orientation. We also consider loops to be a distinct type of orientation, separate from non-loop outgoing and incoming edges. The array, therefore, consists of 5 rows (unmarked, dashed, red, blue, green) and 3 columns (incoming, outgoing, loop), totalling 15 cells that each store a single linked list. See Figure 3 for an illustration.

4 Finding Nodes in Constant Time

In this section, we explain the problem of disconnected input graphs. For example, consider the program `is-discrete` in Figure 4. The program fails if and only if the input graph is discrete, that is, contains no edges. We assume that the input graph is unmarked. The program is composed of a loop followed by a test. The rule `mark` in the loop marks and roots an arbitrary unmarked node while the rule `isolated` checks whether the node rooted by `mark` is isolated. Notice that the node in the left-hand

	in	out	loop
unmarked	...	...	...
dashed	...	...	...
red	...	...	...
green	...	...	...
blue	...	...	...

Figure 3: Two-dimensional array of linked lists of edges.

side of `isolated` is to be deleted and the right-hand side node is to be created. Hence, by the dangling condition, the rule is applicable only to a red root node. If `isolated` is not applicable, the node rooted by `mark` is not isolated and the loop is terminated by the `break` command. Finally, the rule `root` checks if a red root exists in the host graph, which is the case if and only if the application of `isolated` failed.

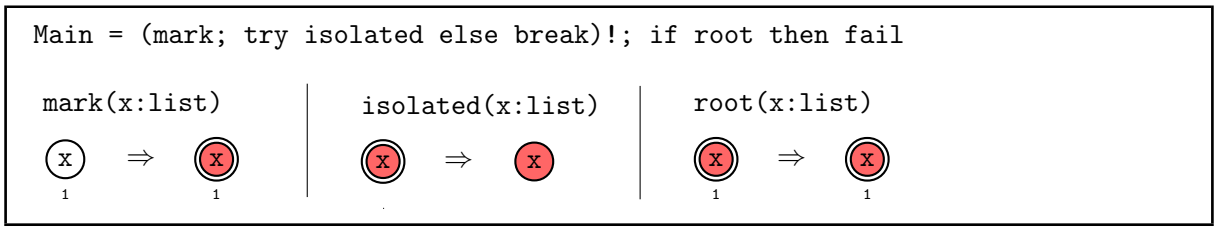


Figure 4: The program `is-discrete`.

The 2020 compiler matched the rule `mark` with a complexity of $O(n)$, where n is the number of nodes in the host graph. In order to find an unmarked node, the matching algorithm had to iterate through the linked list containing all nodes. As the loop body is executed at most n times, the overall complexity of `is-concrete` was $O(n^2)$, as illustrated by the timing diagram in Figure 5.

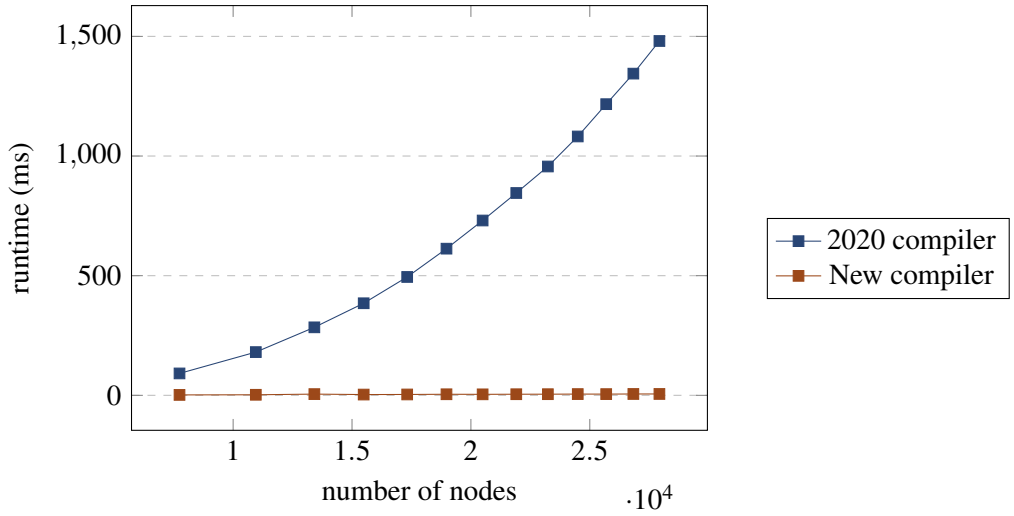


Figure 5: Measured performance of `is-discrete` on discrete graphs under the 2020 compiler and the new compiler.

unmarked	...
grey	...
red	...
green	...
blue	...

Figure 6: Array of linked lists of nodes.

Procedure	Description	Complexity
`alreadyMatched`	Test if the given item has been matched in the host graph.	$O(1)$
`clearMatched`	Clear the `is matched` flag for a given item.	$O(1)$
`setMatched`	Set the `is matched` flag for a given item.	$O(1)$
`firstHostNode(m)`	**Fetch the first node of mark `m` in the host graph.**	$O(1)$
`nextHostNode(m)`	**Given a node of mark `m`, fetch the next node of mark `m` in the host graph.**	$O(1)$
`firstHostRootNode`	Fetch the first root node in the host graph.	$O(1)$
`nextHostRootNode`	Given a root node, fetch the next root node in the host graph.	$O(1)$
`firstInEdge(m)`	**Given a node, fetch the first incoming edge of mark `m`.**	$O(1)$
`nextInEdge(m)`	**Given a node and an edge of mark `m`, fetch the next incoming edge of mark `m`.**	$O(1)$
`firstOutEdge(m)`	**Given a node, fetch the first outgoing edge of mark `m`.**	$O(1)$
`nextOutEdge(m)`	**Given a node and an edge of mark `m`, fetch the next outgoing edge of mark `m`.**	$O(1)$
`firstLoop(m)`	**Given a node, fetch the first loop edge of mark `m`.**	$O(1)$
`nextLoop(m)`	**Given a node and an edge of mark `m`, fetch the next loop edge of mark `m`.**	$O(1)$
`getInDegree`	Given a node, fetch its incoming degree.	$O(1)$
`getOutDegree`	Given a node, fetch its outgoing degree.	$O(1)$
`getMark`	Given a node or edge, fetch its mark.	$O(1)$
`isRooted`	Given a node, determine if it is rooted.	$O(1)$
`getSource`	Given an edge, fetch the source node.	$O(1)$
`getTarget`	Given an edge, fetch the target node.	$O(1)$
`parseInputGraph`	Parse and load the input graph into memory: the host graph.	$O(n)$
`printHostGraph`	Write the current host graph state as output.	$O(n)$

Figure 7: Updated runtime complexity assumptions. Procedures modified in this paper are highlighted in grey. Procedures modified in [2] are highlighted in light blue. n is the size of the input.

5 Second Compiler Enhancement

To overcome the problem described in Section 4, the new graph data structure stores host-graph nodes in five global linked lists. Each list corresponds to one of the node marks red, green, blue, grey or unmarked, and holds all nodes with that mark. The first element of each linked list can be accessed in constant time and hence, for example, the rule `mark` in Figure 4 can be matched in constant time. The matching algorithm inspects the first element in the linked list of unmarked nodes. As the left-hand side of `mark` requires an unmarked node with an arbitrary label, the first node in the list will match. In case the list of unmarked nodes is empty, the host graph does not contain any unmarked node and thus the application of `mark` fails.

As a result of these changes, the time complexity of the program `is-discrete` under the new compiler is reduced to $O(n)$. Figure 5 highlights the difference in runtimes of the program `is-discrete` run under both compilers.

To reason about programs, we need to make assumptions on the complexity of certain elementary

operations. Figure 7 shows the complexity of basic procedures of the C code generated by the GP 2 compiler, adapted from [5]. The grey rows indicate existing procedures updated by the changes introduced in this paper. The time complexities are consistent with the runtimes observed in all our case studies with the new compiler.

6 Case Study: Recognising Acyclic Graphs

Checking whether a given graph contains a directed cycle is a basic problem in the area of graph algorithms [14]. A GP 2 program solving this problem is given in [5], but to run in linear time it requires input graphs of bounded node degree. The same paper contains a program for the related problem of recognising binary DAGs, which are acyclic graphs in which each node has at most two outgoing edges. This program has a linear runtime on arbitrary input graphs but is destructive in that the input graph is partially or totally deleted.

6.1 Program

The program `is-dag` in Figure 8 recognises acylic graphs with respect to the following specification.

Input: An arbitrary GP 2 host graph such that

1. each node is non-rooted and marked grey, and
2. each edge is unmarked.

Output: If the input graph is acyclic, a host graph that is isomorphic to the input graph up to marks. Otherwise *failure*.

Strictly speaking, this program is destructive in case the input graph is cyclic because the `fail` command in the procedure `Check` does not return an output graph. However, `is-dag` could easily be made completely non-destructive by replacing the `fail` command with a rule creating a distinct structure (such as an unmarked node) which signals the existence of a cycle.

Figure 9 illustrates an execution of `is-dag` on a cyclic input graph, and Figure 10, on an acylic graph. The program implements a directed DFS (depth-first search) of the host graph that marks the visited nodes red or blue, where the red nodes are currently being visited. When modelling the DFS with a stack, red nodes are currently on the stack while blue nodes have been previously on the stack and were popped because they require no further visits. Moreover, the top of the stack is a root node.

It is an invariant of `is-dag` that there is at most one root in the host graph throughout the program's execution. The graph contains a cycle if and only if the search finds an edge from the root to a red node. In fact, if u is the root and v is a red node adjacent to u via an edge from u to v, then there must exist a directed path from v to u. Hence a directed cycle has been found.

Consider the loop `(init; DFS!; try unroot else break)!` of `is-dag`'s main procedure. Rule `init` selects an arbitrary grey node as a root to start a directed DFS. The loop `DFS!` moves the root in depth-first fashion through the host graph. The procedure uses a `try-else` command to find any unprocessed (that is, unmarked) edge outgoing from the root. It does this by calling `next_edge`.[3] If there is such an edge, the rule marks it red so that it can be uniquely identified by the rest of the procedure. If no such edge exits, the root can no longer move forward and the `else` statement is invoked instead.

After a successful application of `next_edge`, the root is adjacent to either (1) a grey node, (2) a blue node, or (3) a red node. In case (1), the rule `move` moves the root to the grey node, marks it red and

[3] In the programs of this paper, we use the magenta colour to represent the wildcard mark `any`.

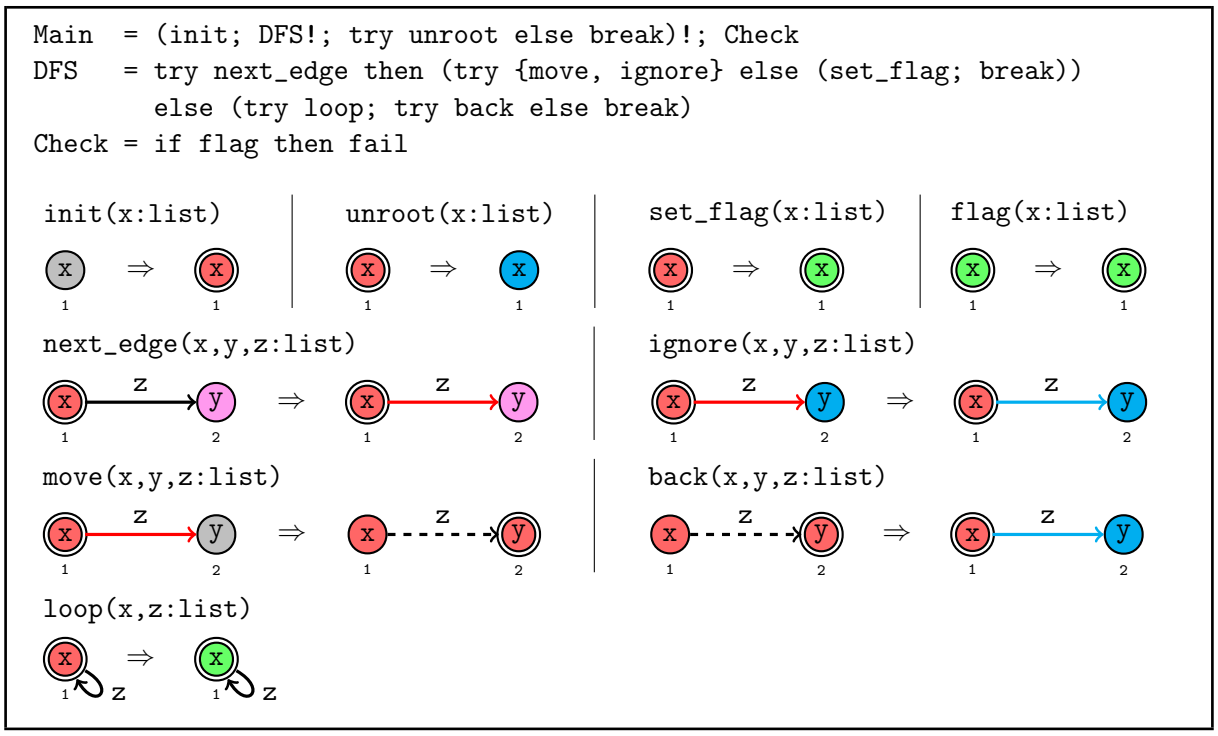


Figure 8: The program `is-dag`.

dashes the traversed edge. Dashed edges represent the path followed by the directed DFS. In case (2), the red edge is marked blue by the rule `ignore` so that it can no longer be matched by `next_edge`. In case (3), neither `move` nor `ignore` is applicable so that `set_flag` marks the root green, indicating the existence of a cycle.

If `next_edge` is not applicable, the command sequence `(try loop; try back else break)` is executed. Rule `loop` checks whether there is a loop attached to the root. If this is the case, the rule marks the root green, similar to `set_flag`. Then rule `back` is tried which implements the *pop* operation in the above mentioned stack model. The rule moves the root backwards along an incoming dashed edge. If no incoming dashed edge is present, the root must be the only element on the stack so that the `break` command terminates the loop `DFS!`.

Upon termination of `DFS!`, the rule `unroot` attempts to turn the root into an unrooted blue node. If this is not possible, the root must have been marked green by `set_flag` or `loop`. This implies the existence of a cycle and hence the outer loop of `is-dag` is terminated.

If rule `unroot` could be applied, there may still be nodes that have not been visited by the DFS. These are nodes that are not directly reachable from the initial nodes chosen so far. In this case the execution of the outer loop is continued until `init` is no longer applicable or `unroot` fails.

Finally, the procedure `Check` tests whether a green flag exists in the host graph. Should this be the case, a cycle was found and the command `fail` terminates the program with failure. Otherwise, the program terminates by returning a host graph which is isomorphic to the input graph up to the generated marks.

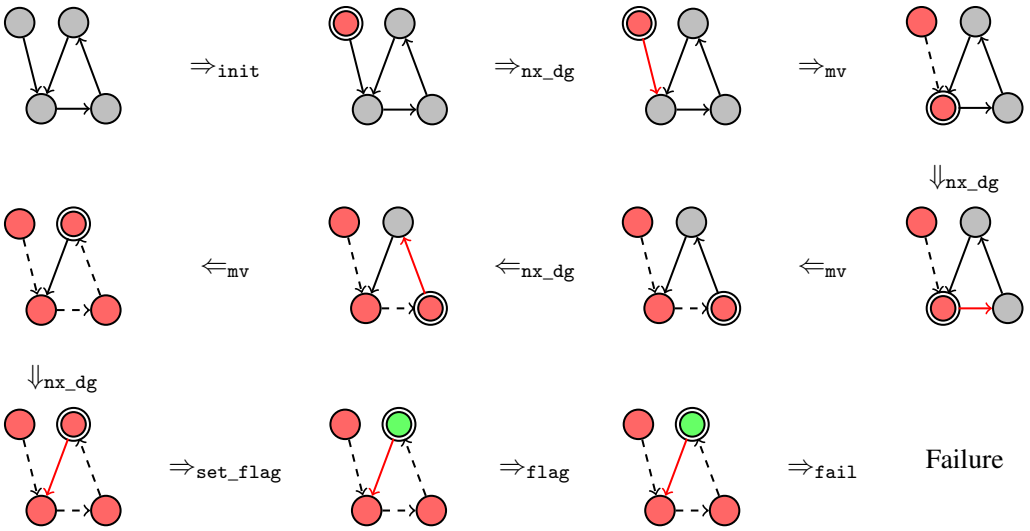


Figure 9: Sample execution of `is-dag` on a cyclic graph.

6.2 Time Complexity

All rules of the program `is-dag` apply in constant time under the complexity assumptions of the modified GP 2 compiler (Figure 7). The rule `init` applies in constant time since any grey-marked node is a match for the rule. As the generated graph data structure keeps a linked list of nodes for each node mark, the matching algorithm can select a grey node in constant time regardless of the number of non-grey nodes in the host graph.

It can be observed that nodes and edges are never remarked by a mark they previously had. Since all rules, except `flag` called at most once in `Check`, remark at least one element, the overall program runtime is linear in the size of the graph, i.e. $O(n)$ where n is the number of nodes and edges in the input graph. To support this claim, we conducted runtime experiments on various classes of bounded-degree graphs (Figures 11, 12, 14, 15 and 16), and unbounded-degree graphs (Figures 13 and 17). The timing results are shown in Figure 18.

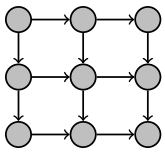


Figure 11: Grid graph.

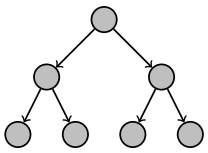


Figure 12: Binary tree.

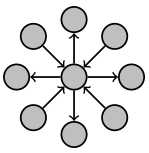


Figure 13: Star graph.

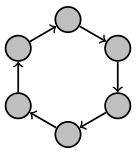


Figure 14: Cycle graph.

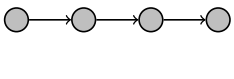


Figure 15: Linked list.

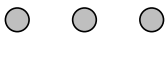


Figure 16: Discrete graph.

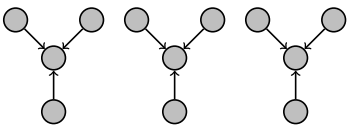


Figure 17: k k-star graphs.

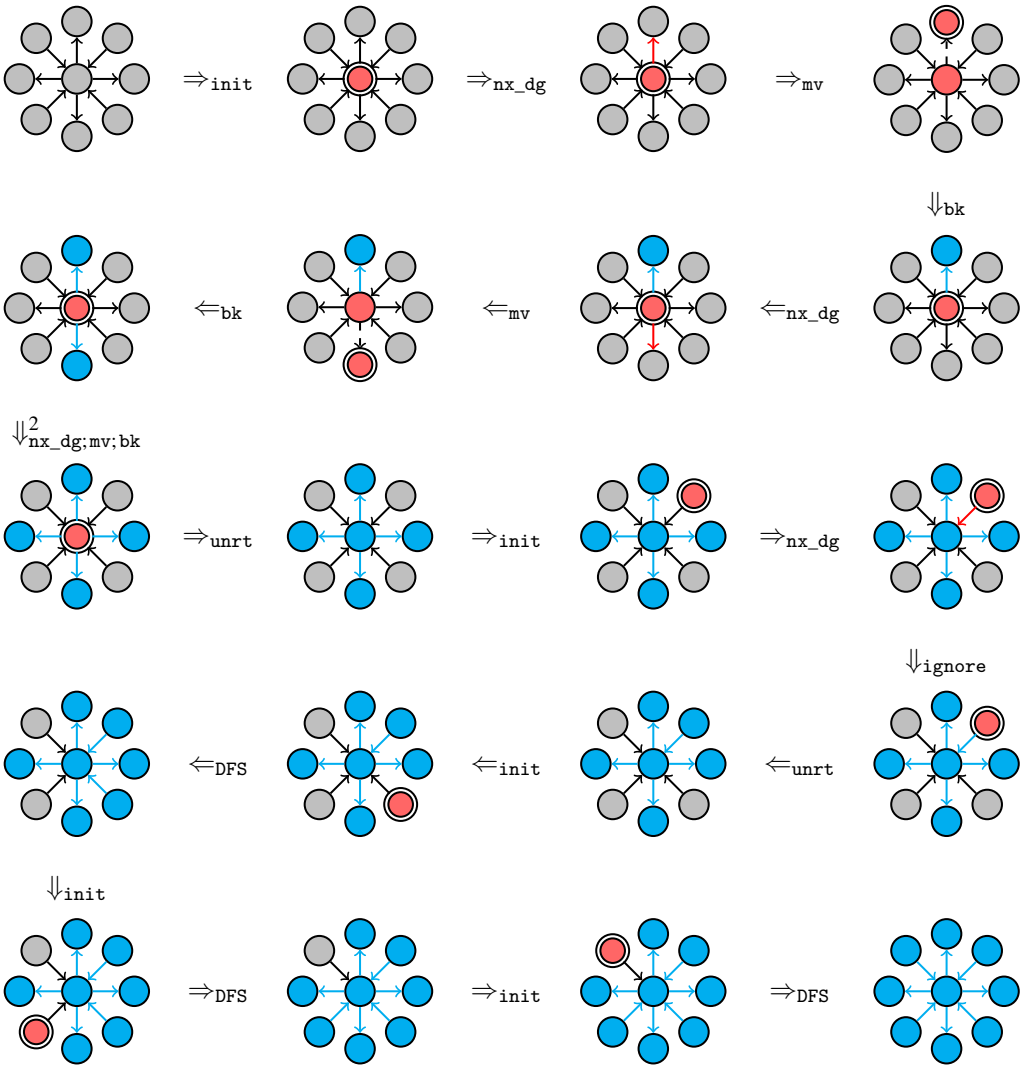


Figure 10: Sample execution of `is-dag` on an acyclic graph.

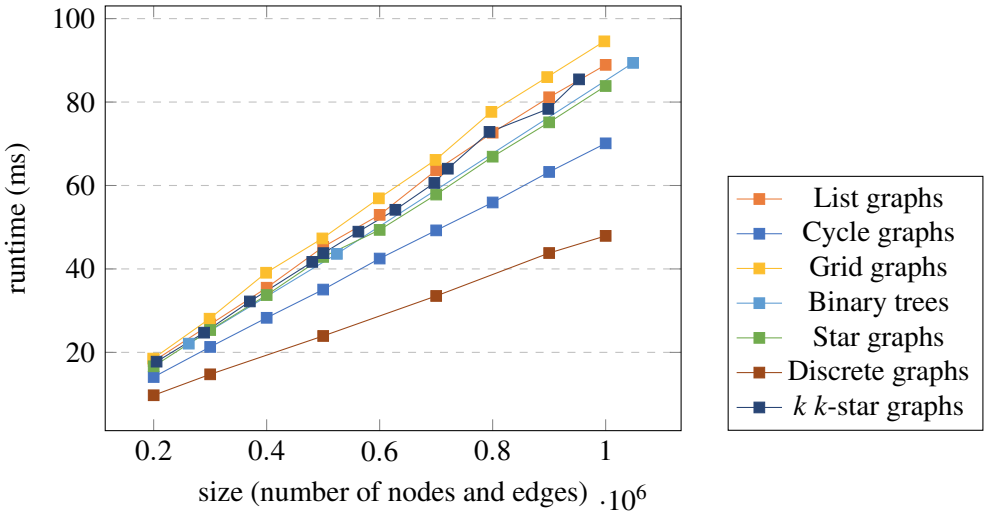


Figure 18: Measured performance of the program `is-dag` under the modified compiler.

7 Case Study: Numbering Connected Components

A clear advantage of the new data structure is the ability to match an arbitrarily-labelled node of a particular mark (or determine that none exists) in constant time. A natural choice of a program that can be constructed under that paradigm is one that numbers all connected components of an input graph.

The program `component-numbering` from Figure 19 appends a number to the list of each node of an input graph unique to the connected component the node belongs to with respect to the following specification.

Input: An arbitrary GP 2 host graph such that

1. each node is non-rooted and marked grey, and
2. each edge is unmarked.

Output: A host graph structually isomorphic to the input graph where a number is appended to the list of each node, denoting the unique identifier of the connected component it belongs to.

7.1 Program

The program `component-numbering` works by first evoking the rule `init`, which marks an arbitrarily-labelled node grey, roots it and appends `1` (first component identifier) to its list label. If the rule fails to match, the graph is empty and the program terminates, given that `unroot` fails. The procedure `DFS` works analogously to that of `is-dag`, except it is undirected. The rule `move` propagates the identifier.

The first looping body `DFS!` propagates the numbering `1` in a single connected component of the graph. The next looping procedure repeats the process, except it invokes `next` instead of `init`. The rule `next` simply unroots the current rooted node, roots another unvisited (grey-marked) node and appends the identifier of the previous rooted node, incremented by 1. Once all unvisited nodes are exhausted, the rule `next` fails to match and the `break` command is called. The rule `unroot` unroots the sole root of the graph.

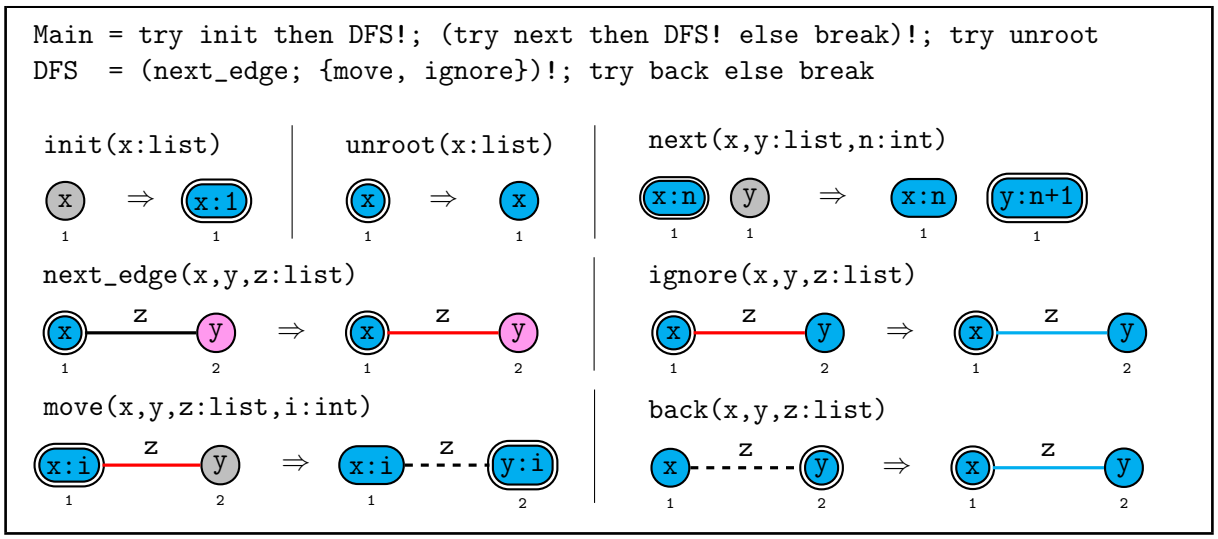


Figure 19: The program `component-numbering`.

7.2 Time Complexity

The time complexity of the program `component-numbering` is linear. That is largely attributed to the fact that `init` and `next` match in constant time, which would have not been possible under the unmodified compiler. Figure 20 offers corroborative evidence. As expected, the program exhibits a linear runtime on discrete graphs, which are disconnected and require $n-1$ calls of the rule `next`, with n being the number of nodes.

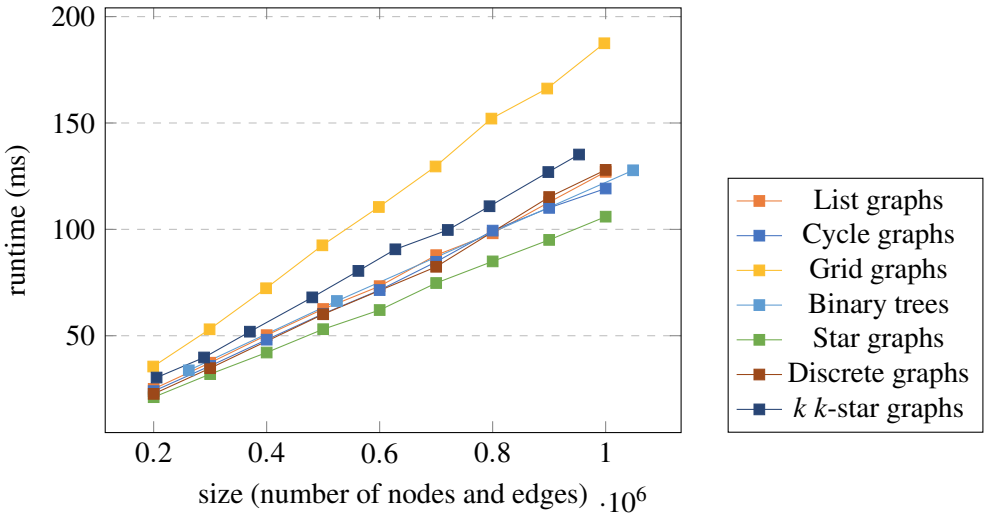


Figure 20: Measured performance of the program `component-numbering` under the modified compiler.

8 Case Study: Breadth-First Search

The problem of traversing a graph in a breadth-first search (BFS) fashion in GP 2 is interesting, as previous techniques used to traverse graphs non-destructively in linear time involved variations of a depth-first search. Bak proposed, in his PhD thesis [3], an implementation of the BFS algorithm in GP 2. However, that program ran in quadratic time. That is due to there being no way, prior to the compiler enhancement of this paper, to find a node to expand from in constant time, given that the next node to expand from is not necessarily adjacent to the one being expanded during a BFS.

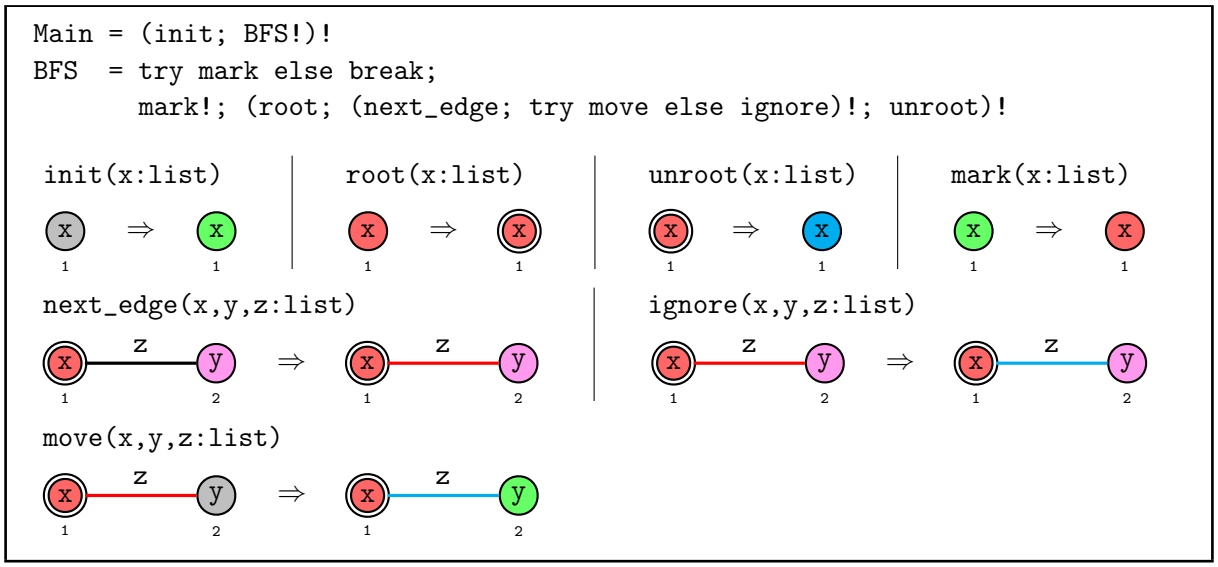


Figure 21: The program `bfs`.

In this section, we present the program `bfs`, capable of carrying out a breadth-first search of a graph in linear time with respect to its size and the following specification.

Input: An arbitrary GP 2 host graph such that

1. each node is non-rooted and marked grey, and
2. each edge is unmarked.

Output: A host graph structually isomorphic to the input graph where all nodes and edges are marked blue.

8.1 Program

The program exploits the advantages of the compiler modifications in order to find the next node to expand from in constant time. The program `bfs` consists of a loop, `(init; BFS!)`, which itself contains a procedure, `BFS`, which marks all green nodes in the host graph red and, subsequently, marks all grey nodes directedly adjacent to the red nodes green. Once a red node has been expanded from, it is marked blue. That procedure loops as long as the connected component of the node marked green by `init` contains non-blue nodes.

Intuitively, at the beginning of the execution of `BFS`, the program seeks to mark all nodes directedly adjacent to the node marked green by `init`. Firstly, the rule `mark` is applied as long as possible to

mark all green-marked nodes red. Then, for as long as possible, the program picks some red node (which was previously green) with the rule `root`, and expands from it as long as possible. The nested looping procedure ends when `next_edge` is no longer applicable, implying that there is no expansion left from the node picked. The rule `unroot` then marks the chosen node blue, indicating that it was fully processed, and moves on to the next red-marked node. The upper parenthetical looping procedure within `BFS!` terminates when `root` no longer applies, indicating that all nodes that were previously green at the beginning of `BFS` have been processed. Once `BFS!` terminates, the rule `init` is called again to start a bread-first search procedure from a different non-visited connected component. The loop continues until all nodes in the host graphs are visited.

8.2 Time Complexity

The program `bfs` runs in linear time with respect to the size of the graph (i.e. the number of nodes and edges).

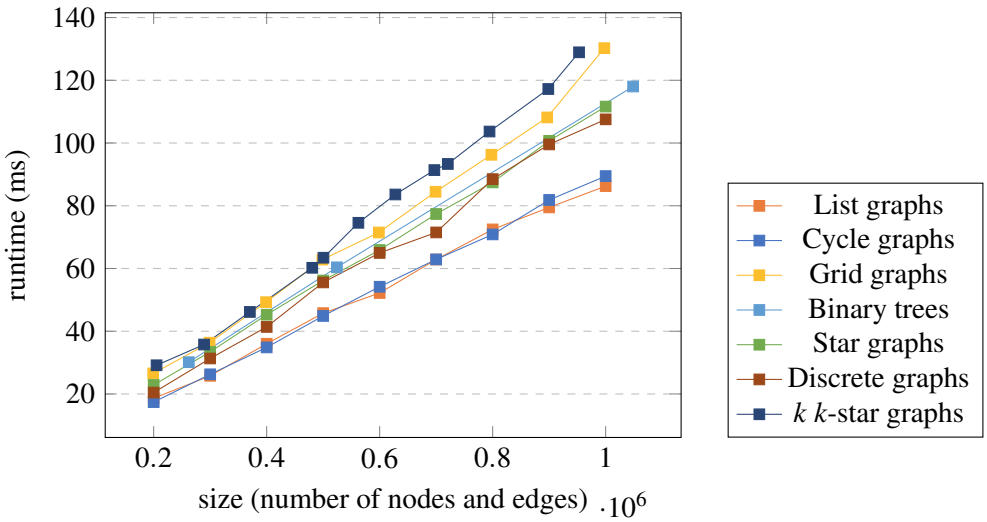


Figure 22: Measured performance of the program `bfs` under the modified compiler.

9 Conclusion

We have demonstrated by case studies how to implement in GP 2 graph algorithms based on depth-first and breadth-first search such that a linear runtime is achieved, even if input graphs have an unbounded node degree or are possibly disconnected. Addressing the issues of unbounded degree and disconnectedness has been an open problem since the publication of the first paper on rooted graph transformation [4]. Up to now, only certain graph reduction programs that destroy their input graphs could be designed to run in linear time on graph classes that have an unbounded node degree or contain disconnected graphs [5].

Our approach involves both enhancing the graph data structure generated by the GP 2 compiler and developing a programming technique that leverages the new graph representation. Previously, the graph data structure in the C program generated by the compiler stored all host-graph nodes in a single linked

list. Hence, if host graph nodes may have different marks, searches within this list required linear time, preventing constant-time rule matching. In contrast, the new data structure finds a node with a given mark or an edge with a given mark and orientation in constant time.

We speculate that all linear-time graph algorithms based on depth-first or breadth-first search can be implemented as GP 2 programs running in linear time. More generally, we intend to implement a GP 2 library of advanced data structures, such as priority queues, Fibonacci heaps, or AVL trees, to support programmers in constructing GP 2 versions of conventional graph algorithms that match the time complexity achievable in the imperative setting.

References

[1] Aditya Agrawal, Gabor Karsai, Sandeep Neema, Feng Shi & Attila Vizhanyo (2006): *The Design of a Language for Model Transformations*. *Software & Systems Modeling* 5(3), pp. 261–288, doi:10.1007/s10270-006-0027-7.

[2] Ziad Ismaili Alaoui & Detlef Plump (2024): *Linear-Time Graph Programs for Unbounded-Degree Graphs*. In: *Proc. 17th International Conference on Graph Transformation (ICGT 2024)*, *Lecture Notes in Computer Science* 14774, Springer, pp. 3–20, doi:10.1007/978-3-031-64285-2_1.

[3] Christopher Bak (2015): *GP 2: Efficient Implementation of a Graph Programming Language*. Ph.D. thesis, Department of Computer Science, University of York, UK. Available at `https://etheses.whiterose.ac.uk/12586/`.

[4] Christopher Bak & Detlef Plump (2012): *Rooted Graph Programs*. In: *Proc. 7th International Workshop on Graph Based Tools (GraBaTs 2012)*, *Electronic Communications of the EASST* 54, doi:10.14279/tuj.eceasst.54.780.

[5] Graham Campbell, Brian Courtehoute & Detlef Plump (2022): *Fast Rule-Based Graph Programs*. *Science of Computer Programming* 214, p. 102727, doi:10.1016/j.scico.2021.102727.

[6] Graham Campbell, Jack Romö & Detlef Plump (2020): *The Improved GP 2 Compiler*. *ArXiv e-prints* arXiv:2010.03993, doi:10.48550/arXiv.2010.03993. 11 pages.

[7] Thomas H. Cormen, Charles E. Leiserson, Robert L. Rivest & Clifford Stein (2022): *Introduction to Algorithms*, fourth edition. The MIT Press.

[8] Heiko Dörr (1995): *Efficient Graph Rewriting and its Implementation*. *Lecture Notes in Computer Science* 922, Springer, doi:10.1007/BFb0031909.

[9] Maribel Fernández, Hélène Kirchner & Bruno Pinaud (2019): *Strategic port graph rewriting: an interactive modelling framework*. *Mathematical Structures in Computer Science* 29(5), pp. 615–662, doi:10.1017/S0960129518000270.

[10] Amir Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon & Maria Zimakova (2012): *Modelling and analysis using GROOVE*. *International Journal on Software Tools for Technology Transfer* 14(1), pp. 15–40, doi:10.1007/s10009-011-0186-x.

[11] Edgar Jakumeit, Sebastian Buchwald & Moritz Kroll (2010): *GrGen.NET – The expressive, convenient and fast graph rewrite system*. *International Journal on Software Tools for Technology Transfer* 12(3–4), pp. 263–271, doi:10.1007/s10009-010-0148-8.

[12] Edgar Jakumeit, Sebastian Buchwald, Dennis Wagelaar, Li Dan, Ábel Hegedüs, Markus Herrmannsdörfer, Tassilo Horn, Elina Kalnina, Christian Krause, Kevin Lano, Arend Rensink, Louis Rose, Sebastian Wätzoldt, Markus Lepper & Steffen Mazanek (2014): *A survey and comparison of transformation tools based on the transformation tool contest*. *Science of Computer Programming* 85, pp. 41–99, doi:10.1016/j.scico.2013.10.009.

[13] Detlef Plump (2012): *The Design of GP 2*. In: *Proc. Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011)*, *Electronic Proceedings in Theoretical Computer Science* 82, pp. 1–16, doi:10.4204/EPTCS.82.1.

[14] Steven S. Skiena (2020): *The Algorithm Design Manual*, third edition. Springer, doi:10.1007/978-3-030-54256-6.

[15] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaela Groner, Timo Kehrer, Manuel Ohrndorf & Matthias Tichy (2017): *Henshin: A Usability-Focused Framework for EMF Model Transformation Development*. In: *Proc. 10th International Conference on Graph Transformation (ICGT 2017)*, *Lecture Notes in Computer Science* 10373, Springer, pp. 196–208, doi:10.1007/978-3-319-61470-0_12.

Grappa RE
A Tool for Efficient Graph Recognition Based on Finite Automata and Regular Expressions

Mattia De Rosa
Department of Informatics
University of Salerno
Fisciano (SA), Italy
matderosa@unisa.it

Mark Minas
Computer Science Department
Universität der Bundeswehr München
Neubiberg, Germany
mark.minas@unibw.de

A recent paper by Drewes, Hoffmann, and Minas (GCM 2023 proceedings) has shown that certain graph languages can be defined and efficiently recognized by finite automata when strings over typed symbols are interpreted as graphs. This approach has been implemented in the tool *Grappa RE*, which is described in this paper. *Grappa RE* allows for the convenient specification of graph languages through regular expressions, converts each of them into a minimized deterministic finite automaton, and checks whether it can recognize graphs without the need for backtracking. Measurements confirm that recognition runs in linear time.

1 Introduction

Engelfriet and Vereijken have described that hypergraphs can be composed of elementary graphs using sequential and parallel composition to define hyperedge replacement grammars [10]. If only sequential composition, i.e., concatenation, is used, then such a composition yields a string over an alphabet (representing elementary graphs). The resulting graphs are therefore interpretations of these strings. Finite automata are a common means of defining regular (string) languages, and by interpreting the generated strings, such automata define graph languages. This idea is not new, finite automata for algebraic structures were already studied by Arbib and Give'on in the 1960s [1, 11], Bozapalidis and Kalampakas analyzed finite automata on Engelfriet's and Vereijken's hypergraphs [4, 16], Brugging, König *et al.* in the context of hypergraphs as cospans [3], and more recently Earnshaw and Sobociński studied regular languages of morphisms in free monoidal categories, which can be interpreted as hypergraphs or string diagrams, with their associated automata [9]. All of these approaches focus on *defining* graph languages through automata.

In contrast, Hoffmann, Drewes, and Minas recently proposed an approach to efficient graph recognition by finite automata in the sense that an automaton is "executed" to decide whether a given graph is a member of its graph language or not [8]. More precisely, given a finite automaton and a graph, a decision procedure can decide whether the graph is a member of the automaton's graph language. This is always possible with an inefficient backtracking algorithm, but for certain automata backtracking can be avoided, allowing a recognition in linear time in the number of graph edges. These automata must satisfy certain conditions, which are presented in [8] together with algorithms for checking them.

This paper describes the tool *Grappa RE*, which implements these checking algorithms, and the recognition procedure for automata satisfying these conditions. Special emphasis is placed on the techniques necessary for efficient graph recognition in linear time. *Grappa RE* also supports regular expressions, which can be used to conveniently specify finite automata. Experiments with this tool confirm

J. Endrullis, D. Grzelak, T. Heindel and J. Kosiol (Eds):
GCM 2023 and GCM 2024
EPTCS 417, 2025, pp. 55–70, doi:10.4204/EPTCS.417.4

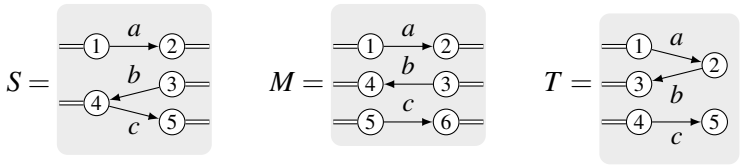


Figure 1: Three graphs S, M, and T.

the statement in [8] that it is possible to recognize graphs with finite automata in linear time. In this way, *Grappa RE* complements the tool landscape of GRAPPA[1]. While GRAPPA (Graph Parsers) is a tool environment for generating efficient top-down and bottom-up parsers for hyperedge replacement grammars and contextual hyperedge replacement grammars [5, 14, 6, 7], *Grappa RE* (GRAPPA for Regular Expressions) realizes graph recognizers for regular expressions.

This paper is organized as follows: The next section describes informally the approach of using finite automata for efficient graph recognition, as proposed in [8], and describes in detail the underlying recognition algorithm with and without backtracking. Then Section 3 describes how this approach has been implemented in *Grappa RE*. Special emphasis is placed on the use of regular expressions as a compact way to specify finite automata, and the efficient implementation of constant time edge selection, which is required for graph recognition in linear time. Section 4 then summarizes the results of experiments with some graph languages specified by regular expressions and recognizable by *Grappa RE*. These results show that *Grappa RE* does indeed recognize graphs in linear time (in the number of their edges). Section 5 concludes the paper.

2 Finite Automata Accepting Graphs

We consider edge-labeled hypergraphs (which we will simply call graphs) without node labels. A ranked vocabulary is used to label hyperedges (which we will simply call edges). Each label ℓ has a rank $rank(\ell) \in \mathbb{N}$, and each edge labeled ℓ is then attached to $rank(\ell)$ different nodes. Each graph G also has a front and a rear interface $front(G)$ and $rear(G)$, respectively, which are repetition-free sequences of nodes in G. Front nodes may also occur in the rear interface. We say that a graph G is of type (m,n) if $|front(G)| = m$ and $|rear(G)| = n$; m is said to be the front type of G and n its rear type.

Figure 1 shows three examples of such graphs with the labels a, b, and c with $rank(a) = rank(b) = rank(c) = 2$. Each graph is contained in a rectangular box, with circles symbolizing nodes and arrows symbolizing binary edges. Nodes of the front interface are marked with double lines starting from the left edge of the box; double lines from the right edge mark nodes of the rear interface. The order of the double lines from top to bottom defines the order of the nodes in that interface. For example, the graph S consists of the nodes 1, 2, 3, 4 and 5 and three edges marked with a, b, and c. The front interface of S consists of the nodes 1 and 4, i.e., the node sequence 14, the rear interface of the node sequence 235. The graphs S, M, and T are of types $(2,3)$, $(3,3)$, and $(3,0)$, respectively, because the rear interface of T is empty.

Two graphs G and H can be concatenated to form a graph $G \odot H$ if their types match, which means that the rear type of G is the same as the front type of H. $G \odot H$ is obtained by taking the disjoint union of G and H and then merging the corresponding nodes from $rear(G)$ and $front(H)$. Figure 2 shows the result of the concatenations $S \odot T$, $S \odot M \odot T$, and $S \odot M \odot M \odot T$. If we define $S \odot M^k \odot T$ as the

[1] GRAPPA and *Grappa RE* are available at https://www.unibw.de/inf2/grappa

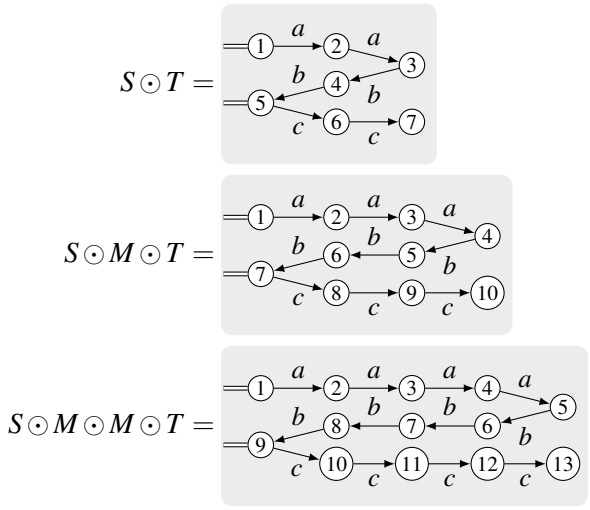


Figure 2: Results of concatenating S, M, and T.

graph resulting from the concatenation of S, k copies of M and finally T, then $S \odot M^k \odot T$ is a graph representation of the string $a^{k+2}b^{k+2}c^{k+2}$.

The approach in [8] is based on the idea of defining a vocabulary of graph symbols and assigning a specific (elementary) graph $[\![s]\!]$ to each symbol s as an interpretation. Strings over this vocabulary can then be interpreted as graphs obtained by concatenating the elementary graphs of the symbols. Of course, this assumes that the string is valid in the sense that it consists only of concatenations of symbols such that the corresponding elementary graphs match in type, so that their concatenation is defined. Otherwise, the string has no valid interpretation.

So-called *blank* and *atom symbols* are defined as vocabulary, with *blanks* and *atoms* as their graph interpretations. Blanks are discrete graphs whose front interface contains all nodes of the blank. In contrast, each atom contains exactly one edge, and no atom has a node that is neither in its front interface nor attached to its edge. Therefore, all nodes of the rear interface must also occur in the front interface, be connected to the edge, or both. These restrictions are necessary for efficient graph recognition, because in this way every node during the recognition process is either already contained in the rear interface of the already recognized subgraph, or is attached to the edge that is read next.

We write blank symbols as $\varepsilon_\rho^{(n)}$ and atom symbols as ℓ_ρ^φ for any $n \in \mathbb{N}$, sequences φ and ρ over $\{1,\dots,n\}$ without repetitions, and edge label ℓ with $rank(\ell) \leq n$. The corresponding blank and atom, $[\![\varepsilon_\rho^{(n)}]\!]$ and $[\![\ell_\rho^\varphi]\!]$, respectively, have nodes $1,\dots,n$ and ρ as their rear interfaces. $[\![\varepsilon_\rho^{(n)}]\!]$ has $1\cdots n$, $[\![\ell_\rho^\varphi]\!]$ has φ as its front interface. The unique edge of $[\![\ell_\rho^\varphi]\!]$ is labeled ℓ and attached to the nodes $1,\dots,rank(\ell)$ (in that order).

For example, Figure 3 shows that the graph S in Figure 1 is isomorphic to a concatenation of the three atoms $[\![a_{23}^{13}]\!]$, $[\![b_{312}^{32}]\!]$, and $[\![c_{342}^{341}]\!]$, i.e., S is an interpretation of $a_{23}^{13}\, b_{312}^{32}\, c_{342}^{341}$. Similarly, M and T are interpretations of $a_{234}^{134}\, b_{314}^{324}\, c_{342}^{341}$ and $a_{234}^{134}\, b_{3}^{123}\, c_{\varepsilon}^{1}$, respectively.

A finite automaton over such symbols defines a language of strings accepted by the automaton, and thus defines a graph language, as long as all these strings have valid graph interpretations, which is easy to ensure: We define the type of each symbol to be the type of its elementary graph ($\varepsilon_\rho^{(n)}$ and ℓ_ρ^φ then have type $(n,|\rho|)$ and $(|\varphi|,|\rho|)$, respectively), and consider only those automata that allow assigning a rank

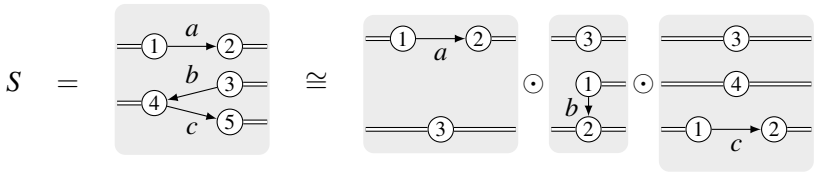


Figure 3: S as in Figure 1 as an interpretation of $a^{13}_{23}\, b^{32}_{312}\, c^{341}_{342}$.

$rank(q) \in \mathbb{N}$ to each state q, such that each incoming transition of q has a symbol of rear type $rank(q)$, and each outgoing transition of q has a symbol of front type $rank(q)$.

Let $a^n b^n c^n$ denote the language of all graphs representing strings in $\{a^k b^k c^k \mid k \in \mathbb{N}\}$. Continuing the example above, Figure 4 shows a finite automaton accepting all strings that can be interpreted as graphs in $a^n b^n c^n$. q_0 is its initial state, q_5 its only final state. The states have the ranks $rank(q_0) = rank(q_1) = 2$, $rank(q_2) = rank(q_4) = rank(q_6) = 3$, $rank(q_3) = 1$, and $rank(q_5) = 0$.

2.1 Recognition with Backtracking

While interpreting a string accepted by an automaton is straightforward, recognizing a given graph, i.e., finding a string representation of the graph such that the string is accepted by the automaton, involves a search process, usually with backtracking due to nondeterministic decisions. Algorithm 1 outlines this process as a classic depth-first search with backtracking using the recursive *depthFirst* procedure. It tries to find a walk through the finite automaton from the initial state to some final state that accepts a string with the input graph G as its interpretation. It either fails in Line 6 or terminates successfully in Line 8.

The algorithm does not call the *depthFirst* procedure but fails immediately if G has a discrete node that is not in its front interface (Line 5 and Line 6). Recall that no atom or blank can create a discrete node, since all nodes in the rear interface also occur in the front interface or are attached to the (unique) edge of the atom.

The *depthFirst* procedure is called with three parameters q, F, U: q is a state of the automaton, F (for "front") contains a sequence of nodes that is the front interface of the next elementary graph to be read, and U (for "unread") contains all edges of G that have not yet been read. The procedure finally terminates successfully if it can (recursively) find a walk from q to some final state such that it reads all edges in U, starting with F as the front interface and ending with the rear interface of G. This means that if *depthFirst* is called with q being a final state, F being the rear interface of G, and $U = \varnothing$, Algorithm 1 will terminate successfully (Line 8). Otherwise *depthFirst* tries each of the outgoing transitions of q in the usual depth-first manner in Line 9, i.e., it selects an arbitrary transition first and continues the search with recursive procedure calls to *depthFirst* (see below). These calls return only if they cannot find a

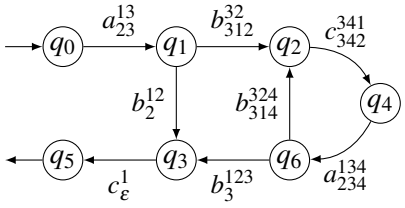


Figure 4: Finite automaton accepting graphs in $a^n b^n c^n$.

Algorithm 1: Recognizing a graph with a finite automaton using backtracking.

Input : input graph G and a finite automaton $\mathfrak{A}$.
Output: *success* if $\mathfrak{A}$ accepts G, and *failure* otherwise.

```
1  begin
2      q ← initial state of 𝔄;
3      F ← front interface of G;
4      U ← all edges of G;
5      if all discrete nodes of G occur in F then call depthFirst(q, F, U);
6      stop with failure

7  procedure depthFirst(q: State, F: Sequence of nodes, U: Set of edges)
8      if q is final, F is the rear interface of G, and U = ∅ then stop with success;
9      foreach outgoing transition t of q do
10         let q′ be the state reached by t;
11         if t is an atom transition then
12             let α be the atom symbol of t;
13             foreach edge e ∈ U as specified by α and attached to nodes in F do
14                 let F′ be F after modifying it according to α and e;
15                 call depthFirst(q′, F′, U \ {e})
16         else
17             let F′ be F after modifying it according to the blank symbol of t;
18             call depthFirst(q′, F′, U)
```

successful walk. The next transition is then selected in Line 9, and so on. The current *depthFirst* call is terminated, i.e., Algorithm 1 backtracks, if all outgoing transitions of q lead to a dead end.

If the selected transition in Line 9 is an atom transition, i.e, if it is labeled with an atom symbol α, it tries to find a matching edge e not yet read (Line 13); α and F determine which edge may be selected: α determines the label of e and how it must be connected to nodes in F. In particular, the front nodes of $[\![\alpha]\!]$ must correspond to the nodes in F. Nodes that are not in F but are attached to e must be "new" nodes, i.e., they must not have been encountered before. The latter follows from the fact that nodes of $[\![\alpha]\!]$ that do not appear in its front interface are created by this step. If the recognizer has several candidates to choose from in Line 13, it will select them one by one by recursively calling *depthFirst*. If this call fails, it will try the next one, and so on.

However, if the selected transition in Line 9 is labeled with a blank symbol, no further decision is required; a blank can always be processed, it just modifies the rear interface of the subgraph of G read so far (Line 17) and continues the depth-first search by recursively calling *depthFirst* in Line 18.[2]

2.2 Recognition without Backtracking

This search can be expensive because it uses backtracking to try out possible solutions instead of systematically selecting suitable transitions and edges, which can lead to exponential runtimes. To address this

[2]Note that a cycle of blank transitions can lead to an infinite recursion, which could easily be prevented by keeping track of the transitions visited so far. However, this has been omitted here to simplify the presentation.

Algorithm 2: Recognizing a graph with a DFA that has the FEC and TS property.

Input : input graph G and a DFA $\mathfrak{A}_d$ that has the FEC and TS property.
Output: *success* if $\mathfrak{A}_d$ accepts G, and *failure* otherwise.

```
1  if G contains discrete nodes that are not in its front interface then stop with failure;
2  q ← initial state of 𝔄_d;
3  F ← front interface of G;
4  U ← all edges of G;
5  while U ≠ ∅ do
6      foreach outgoing atom transition t of q do
7          let α be the atom symbol of t;
8          if there is an edge e ∈ U as specified by α and attached to nodes in F then
9              let e be such an edge;
10             modify F according to α and e;
11             set q to the state reached by t;
12             remove e from U;
13             continue while loop at Line 5
14     stop with failure
15 if q is a final state and F is the rear interface of G then stop with success;
16 if there is an outgoing blank transition of q such that its blank turns F into the rear interface of G
       then stop with success;
17 stop with failure
```

problem, [8] has discussed conditions under which backtracking can be avoided completely in this search process. In fact, the automaton must satisfy additional conditions to allow a deterministic recognition process without backtracking: It must have the *transition selection* (TS) property and the *free edge choice* (FEC) property, which will be explained below. Since backtracking is then not necessary, Algorithm 1 is transformed into a simplified version (Algorithm 2), which will be discussed next.

It is clear that a deterministic choice of a suitable transition is impossible if the automaton $\mathfrak{A}$ is nondeterministic. That is, if it contains a state with two outgoing transitions labeled with the same graph symbol. Therefore, the finite automaton must first be transformed into a deterministic finite automaton (DFA), as in the case of transforming a nondeterministic finite automaton for strings, before it can be used by Algorithm 2. The approach proposed in [8] uses a modified version of the well-known powerset construction. Surprisingly, however, the powerset construction does not always produce a deterministic automaton in this context. This can happen if $\mathfrak{A}$ uses atom symbols which differ only in their rear interfaces (see [8, Example 4.1]). In this case, however, one can always transform $\mathfrak{A}$ into an equivalent automaton $\mathfrak{A}'$, which can then be transformed into an equivalent DFA $\mathfrak{A}_d$ using the modified powerset construction. The transformation of $\mathfrak{A}$ into the DFA $\mathfrak{A}_d$ is thus a two-step process, consisting of the preprocessing step (called *disambiguation*) and the following (modified) powerset construction.

If $\mathfrak{A}_d$ produced by the modified powerset construction contains blank transitions, all of them reach a final state that has no outgoing transitions. Consequently, a blank transition can only be selected if all edges of G are have been read. All previously selected transitions must be atom transitions. Each loop cycle of the while loop (Lines 5–14 of Algorithm 2) selects an atom transition and then reads a corresponding edge of G. The TS property of $\mathfrak{A}_d$ implies that the only possible transition can be selected

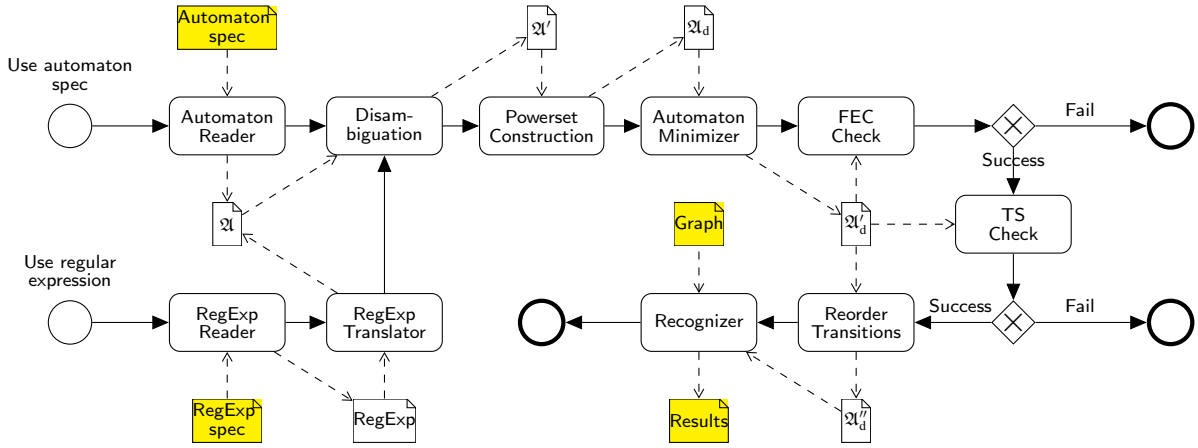


Figure 5: Architecture of *Grappa RE*

as shown in Algorithm 2: The foreach loop (Lines 6–13) tries all outgoing atom transitions of the current state in a predefined order until it finds the first one such that G contains a matching edge that has not yet been read before. This order is determined by the algorithm that checks the TS property of $\mathfrak{A}_\mathrm{d}$ described in [8]. In fact, $\mathfrak{A}_\mathrm{d}$ has the TS property if such an order exists for each of its states.

Note that Algorithm 2 in Line 9 selects any unread edge e that matches the atom symbol α of the selected transition. In fact, every edge that matches α is equally suitable if $\mathfrak{A}_\mathrm{d}$ has the FEC property: If one of these candidates is the right choice for $\mathfrak{A}_\mathrm{d}$ to accept G, then all of them are. [8] has proposed a sufficient criterion for checking the FEC property, i.e., $\mathfrak{A}_\mathrm{d}$ has the FEC property if the criterion is met. But if the criterion is not met, $\mathfrak{A}_\mathrm{d}$ may or may not have the FEC property. The criterion consists of identifying all transitions in $\mathfrak{A}_\mathrm{d}$ that are *deferrable*. These are transitions where Algorithm 2 can choose between several possible candidates in Line 9. They are called deferrable because one of the candidates is selected now, while the others must be selected in later cycles of the while loop, possibly using other transitions. [8] has outlined how to identify deferrable transitions. The criterion now requires that the rear interface of α does not contain any nodes that are not also contained in its front interface, i.e., the selection of e in Line 9 cannot affect the selection of F in Line 10.

Algorithm 2 terminates successfully after it has read all edges of G in its while loop and has either reached a final state where F and the rear interface of G are identical (Line 15), or it can reach a final state by taking a blank transition that modifies F in the rear interface of G (Line 16). Otherwise, $\mathfrak{A}$ does not accept G and Algorithm 2 terminates with a failure.

As noted above, a single edge is read in each cycle of the while loop in Lines 5–14. [8] argued that a loop cycle takes only constant time if edge selection is implemented properly. Consequently, Algorithm 2 takes only linear time (in the number of edges) to recognize an input graph. This is demonstrated in Section 4.

3 *Grappa RE*

The approach described in [8] and summarized in the previous section has been realized in the tool *Grappa RE*. It is implemented in JAVA and uses several external packages, in particular `dk.brics.automaton` [17] with its standard algorithms for finite automata and regular expressions for strings, and

GRAPHSTREAM [2] for graph visualization. The architecture of *Grappa RE* is represented as a BPMN[3] diagram in Figure 5. As usual, rectangles represent activities, document icons represent data, either external files (e.g., *Automaton spec*) read or written by *Grappa RE*, which have a yellow background, or internal data (e.g., $\mathfrak{A}$), drawn with a white background. The gateways represent decisions where only one of the outgoing paths is taken.

Grappa RE can be used in two modes, represented by the two different start events, i.e., either with the specification of a finite automaton ("*Use automaton spec*"), or with a regular expression ("*Use regular expression*"). We will describe the first mode first, then Section 3.1 will describe the second mode. Both modes can be used interactively with a GUI or in a headless variant. The GUI is described in Section 3.3, and the headless variant was used in the experiments to check the speed of the recognition procedure described in Section 4.

Grappa RE reads finite automata and input graphs from text files. In the first mode ("*Use automaton spec*"), the *Automaton Reader* reads the specification of a finite automaton from a text file. Figure 6 shows such a textual specification of the automaton in Figure 4. The specification must contain the ranked alphabet of edge labels after the `symbol` keyword; the rank of each symbol is given in parentheses. The states are listed after the `state` keyword. State ranks (see Section 2) are again given in parentheses. Final states are marked with `*`, the start state with the `start` keyword, and transitions in the the obvious way. For example, `a^13_23` represents the atom symbol a_{23}^{13} and `c^1_<>` represents the atom symbol c_{ε}^{1}. Blank symbols such as $\varepsilon_{12}^{(2)}$ are written as `<>^2_12`, but do not appear in Figure 6.

The *Automaton Reader* produces an internal representation $\mathfrak{A}$ of the automaton and then transforms it into a DFA in the two-step process outlined in Section 2: *Disambiguation* is the preprocessing step that ensures that the *Powerset Construction* really produces a deterministic finite automaton, denoted here by $\mathfrak{A}_\mathrm{d}$.

Similar to the string case, the DFA $\mathfrak{A}_\mathrm{d}$ can contain nondistinguishable states. These are states that accept the same languages over graph symbols when starting in these states. Since sequences of graph symbols are special cases of strings, one can use any standard algorithm to minimize $\mathfrak{A}_\mathrm{d}$ by merging all non-distinguishable states. *Grappa RE* uses Hopcroft's algorithm [15] with an implementation provided by `dk.brics.automaton` [17], which yields an equivalent DFA $\mathfrak{A}'_\mathrm{d}$.

The following steps in Figure 5 check if backtracking can be avoided when using $\mathfrak{A}'_\mathrm{d}$ to recognize graphs. *FEC Check* checks if it has the FEC property (see Section 2), and *TS Check* checks if it has the TS property using the algorithms described in [8]. If one of them fails, efficient graph recognition cannot be guaranteed, and *Grappa RE* refuses to use $\mathfrak{A}'_\mathrm{d}$ for graph recognition. However, if both succeed, *Reorder Transitions* then reorders the transitions of each state as required by Algorithm 2 and described

[3] Business Process Management Notation

```
auto abc {
  symbol a(2), b(2), c(2);
  state q0(2), q1(2), q2(3), q3(1), q4(3), q5(0)*, q6(3);
  start q0;
  q0 -- a^13_23   --> q1;  q1 -- b^32_312 --> q2;  q1 -- b^12_2    --> q3;
  q2 -- c^341_342 --> q4;  q3 -- c^1_<>   --> q5;  q4 -- a^134_234 --> q6;
  q6 -- b^324_314 --> q2;  q6 -- b^123_3  --> q3;
}
```

Figure 6: *Grappa RE* specification of the automaton in Figure 4.

```
regexp abc {
  symbol a(2), b(2), c(2);

  a^13_23 b^12_2 c^1_<>
  |
  a^13_23 b^32_312 c^341_342
  (a^134_234 b^324_314 c^341_342)*
  a^134_234 b^123_3 c^1_<>
}
```

Figure 7: Regular expression for the automaton in Figure 4.

in [8]. In fact, this is done by topologically sorting the (atom) transitions of each state according to a partial order obtained by the TS check. The output of this step is a DFA $\mathfrak{A}''_{\mathrm{d}}$.

$\mathfrak{A}''_{\mathrm{d}}$ is finally used in the *Recognizer* step that implements Algorithm 2. The input graph G is either read from an external file in JSON format, or it can be generated programmatically in headless mode. This step checks if $\mathfrak{A}''_{\mathrm{d}}$ (and therefore $\mathfrak{A}$) accepts G by finding a sequence s of graph symbols such that $G \cong [\![s]\!]$. The result of this step, which can be visually inspected in GUI mode, is described in Section 3.3.

The *Recognizer* step tries to recognize the input graph as described by Algorithm 2, i.e., by selecting suitable edges of G step by step. *Grappa RE* provides two different implementations of this edge selection: a simple implementation searches for a suitable edge iteratively and thus takes linear time (in the number of edges of G). Graph recognition then takes quadratic time, as shown in Section 4. However, *Grappa RE* also provides a more efficient selection implementation, described in Section 3.2, which takes only constant time. Thus, *Grappa RE* can recognize graphs in linear time, as claimed in [8] and demonstrated in Section 4.

3.1 Regular Expressions

Grappa RE also provides a second mode where the finite automaton is not specified directly as in Figure 6, but by a regular expression. These regular expressions use graph symbols as elementary symbols, but are otherwise structured like normal regular expressions. In particular, they use concatenation and alternatives, as well as the Kleene star, with the same semantics as in the string case. Each such expression corresponds in the usual way to a finite automaton with the same language. However, not every regular expression over graph symbols is a valid expression. Rather, the finite automaton corresponding to the expression may only accept valid graph interpretations as sequences of graph symbols, i.e., the types of successive graph symbols must match. Just as this can be ensured for automata (see Section 2), it is also possible for regular expressions, as long as it is not just ε, because it does not have any type information, i.e., how many front and rear nodes a corresponding graph has. Consequently, ε is not a valid regular expression.

For example, the automaton shown in Figure 4 and Figure 6 can be more compactly described by the regular expression

$$a_{23}^{13}\, b_{2}^{12}\, c_{\varepsilon}^{1} \;\Big|\; a_{23}^{13}\, b_{312}^{32}\, c_{342}^{341}\, (a_{234}^{134}\, b_{314}^{324}\, c_{342}^{341})^{*}\, a_{234}^{134}\, b_{3}^{123} c_{\varepsilon}^{1}.$$

Step *RegExp Reader* (see Figure 5) can read such a regular expression in textual format as in Figure 7. This specification must also include a declaration of the ranked vocabulary of edge labels. Graph symbols are then specified with the same syntax as in automata specifications (e.g., Figure 6).

RegExp Reader also checks if the regular expression is valid, and *RegExp Translator* then transforms it into an equivalent finite automaton, yielding $\mathfrak{A}$ as an internal representation of the resulting finite automaton. To implement this step, it again uses `dk.brics.automaton` [17], which provides an implementation for the string case that can be used directly here. The automaton $\mathfrak{A}$ is then used in the same way as in the mode "*Use automaton spec*".

3.2 Efficient Edge Selection

As described above, *Grappa RE* implements edge selection as used in Line 9 of Algorithm 2 in a straightforward simple way and in an efficient proper way. We will refer to them as the *simple* and *efficient* implementations, respectively.

To speed up edge selection, both implementations build a data structure before starting the recognition process. The simple implementation simply collects lists of edges, where each list contains all edges with the same label. When Line 9 needs to select an edge as specified by an atom symbol ℓ_ρ^φ, it inspects the list for ℓ and looks for the first edge attached to the current nodes in F as specified by φ. Once a matching edge is found, it can be efficiently removed from the list because it is implemented as a doubly linked list. However, the sequential search for this edge takes linear time in the number of edges with the label ℓ. Section 4 shows that graph recognition with this simple implementation then takes quadratic time and is rather slow.

The efficient implementation builds more sophisticated data structures before starting the recognition process and uses techniques similar to those in GRAPPA for efficient graph parsing [14]. It uses a two-step process:

1. In the first step, the automaton $\mathfrak{A}''_{\mathrm{d}}$ (see Figure 5) is analyzed and all atom symbols in $\mathfrak{A}''_{\mathrm{d}}$ are collected in a set. This set contains all atom symbols that can specify an edge selection (as in Line 9 in Algorithm 2). Each of the atom symbols ℓ_ρ^φ specifies how the edge to be selected must be connected to the current nodes in F. The other nodes of the edge, i.e., those that are not referred to by φ, must be nodes that have not been encountered before. This information is then used in the second step to collect suitable lists of edges, which speeds up edge selection.
2. In the second step, lists of edges of the input graph G are collected, one list for each situation that can occur during edge selection (as in Line 9 in Algorithm 2). Each of these lists will contain all edges that are suitable in the corresponding situation.

As an example, consider the graph language $a^n b^n c^n$ and its DFA in Figure 4. It contains the atom symbols a_{23}^{13}, a_{234}^{134}, b_2^{12}, b_3^{123}, b_{312}^{32}, b_{314}^{324}, c_ε^1, and c_{342}^{341}. Now consider the situation of selecting an edge that matches $\ell_\rho^\varphi = a_{23}^{13}$, F in Algorithm 2 must consist of two nodes, say $F = (m_1, m_2)$. 1 at position 1 of $\varphi = 13$ indicates that one must choose an a-edge attached to two nodes (n_1, n_2) such that $n_1 = m_1$, and n_2 is not yet read since 2 does not occur in φ. Finding such an edge is easy and takes only constant time if each node has a list of all a-edges connected to that node as its first node. This is the usual association list approach. Note, however, that one cannot select an a-edge whose second node has already been read. Consequently, one must remove all a-edges from the corresponding association lists as soon as their second attached node has been encountered. This can also be done efficiently if each edge keeps the information in which association lists it is stored. Their number has an upper bound, which can be read from the set of atom symbols of $\mathfrak{A}''_{\mathrm{d}}$. And removing edges from all these lists can also be done efficiently if these lists are doubly linked lists and one keeps references to the corresponding list buckets. Therefore, removing an edge from all corresponding lists also takes constant time.

However, keeping only simple association lists is not always sufficient to ensure constant time edge selection. To see this, consider the *Spikes* graph language represented by

$$s^{124}_{134}\,(s^{134}_{124})^*\,s^{123}_{\varepsilon} \;\Big|\; s^{423}_{421}\,(s^{421}_{423})^*\,s^{321}_{\varepsilon} \;\Big|\; s^{143}_{243}\,(s^{243}_{143})^*\,s^{123}_{\varepsilon}$$

where label s has rank 3. We call these graphs *Spikes* because of their shape; Figure 8 shows three of them. Nodes are drawn as circles, s-edges as rectangles connected to their attached nodes. Numbers indicate the order of the attachments. *Spikes* graphs are of type $(3,0)$. Nodes of their front interfaces are indicated by lines to the left border, ordered from top to bottom. Graph recognition must start with a unique edge; for the graphs in Figure 8 this is always the edge attached to n_1, n_2, n_3. The central node n_1, which is connected to all edges of the graph, can be any node of its front interface, as shown in Figure 8. Consequently, any front node can be attached to any number of edges. Thus, it is impossible to select the unique starting edge in constant time by simply accessing an association list of a front node. Instead, one must be able to look up edges by pairs of their nodes, as can be seen in the following example: The first alternative of the regular expression starts with s^{124}_{134}, i.e., we are looking for an s-edge whose first and second attached nodes are the first and second nodes in its front interface, i.e., n_1 and n_2 for the left *Spikes* graph. This is only the case for the edge attached to n_1, n_2, n_3.

Grappa RE uses hash tables to manage association lists for tuples of nodes and to allow efficient edge lookup. The association lists are created and filled with edges before recognition starts. The hash table lookup, and thus the association list lookup, takes constant time on average because each hash table is fixed after preprocessing; only the contents of the association lists are modified in the same way as described above. Since the number of atom symbols occurring in a DFA is fixed, all these data structures require linear space in the size of the input graph if each hash table size is chosen to be proportional to the number of all edges. In addition, on average, setting up these data structures takes linear time in the size of the input graph.

3.3 Graphical User Interface

Grappa RE also provides a GUI that allows the user to interactively visualize the recognition process. Specifically, as shown in Figure 9, the *Grappa RE* window is divided into three parts: on the top left is a visualization of the input graph, on the top right is the visualization of the minimized DFA and a bar to control execution, and at the bottom is the sequence of atoms recognized during execution. Since

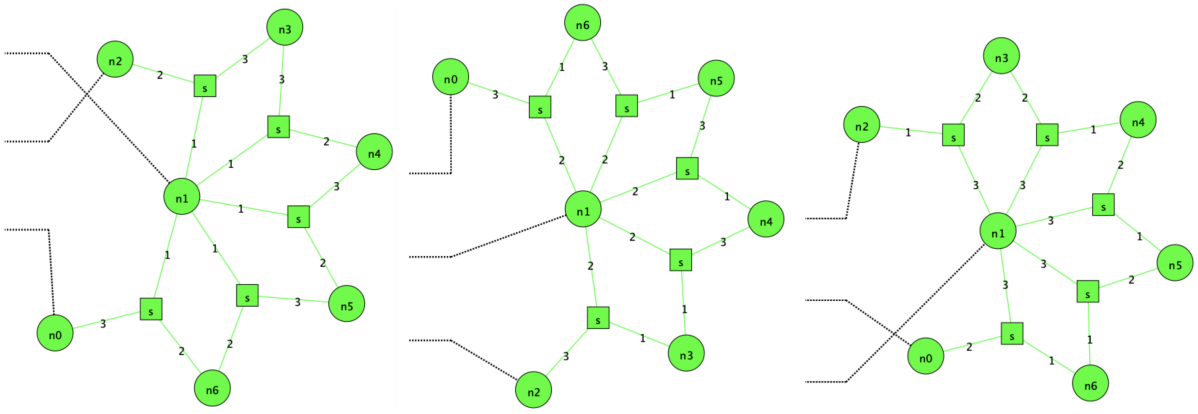


Figure 8: Three *Spikes* graphs.

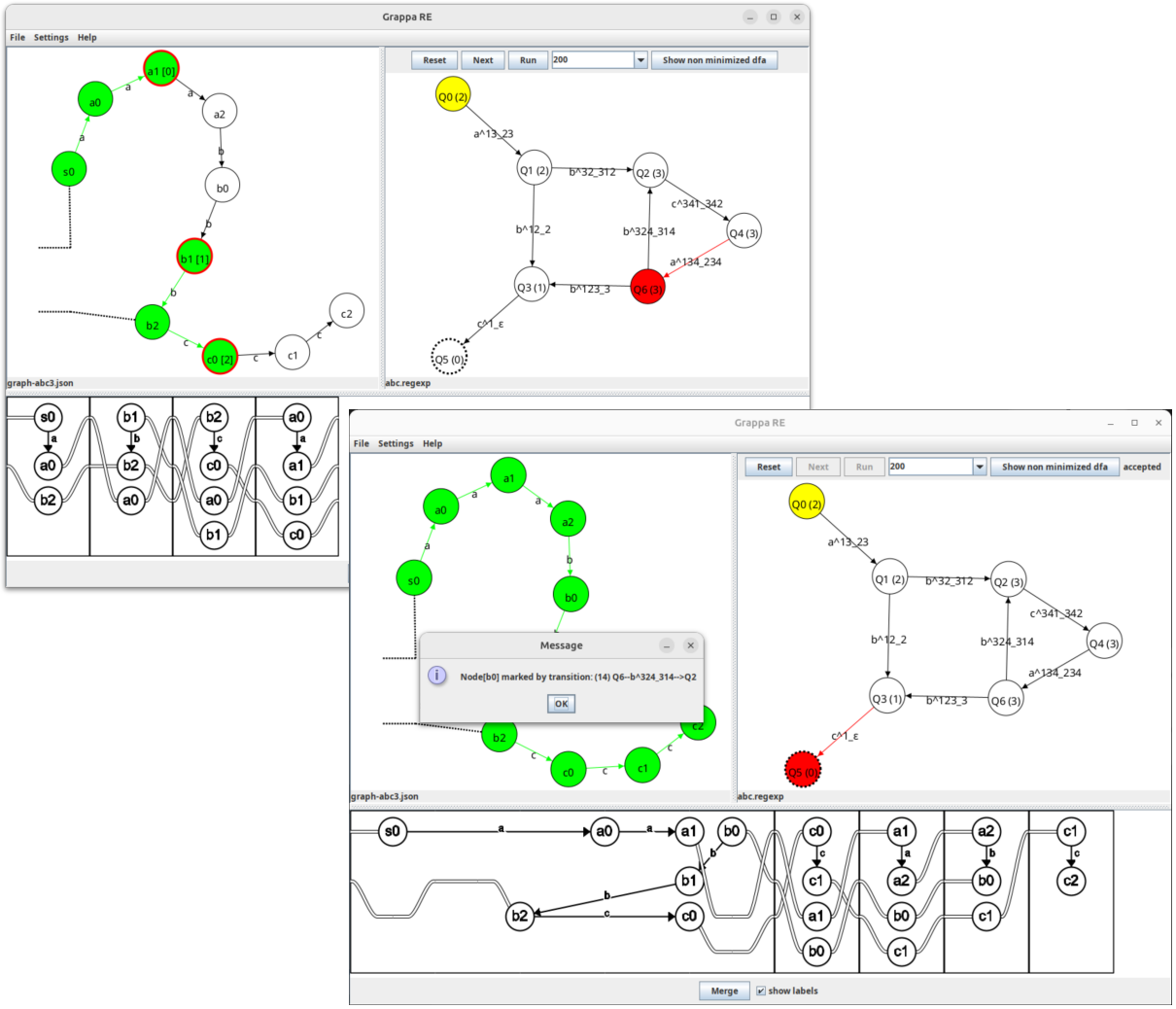


Figure 9: Two screenshots of *Grappa RE* with a graph representing *aaabbbccc* and the automaton recognizing $a^n b^n c^n$ (see also Figure 4). Top: The current state sequence: Q0, Q1, Q2, Q4 and Q6. Lower: Completed execution (accepted graph). A dialog tells you which transition has read and marked a node by clicking on it, here *b0*. At the bottom you can see the result after the user merged the first four atoms by clicking the *Merge* button four times.

the input graph does not provide any layout information, a force-directed layout algorithm [2] is used to display it, which also allows the user to manually move nodes by dragging them with the mouse. The same type of interface was used for the DFA, where the initial state is highlighted in yellow and accepting states have a dashed border instead.

The user can advance the execution of the DFA manually step by step by pressing the *Next* button, or automatically at a selected time interval by pressing the *Run* button. During this operation, the current state of the DFA is highlighted in red, as well as the edges representing the transitions when they are followed. "Consumed" input graph elements are also highlighted (in green) during execution, and it is also possible to click on them later to get information about which transition marked them (as shown in

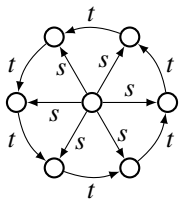


Figure 10: A wheel graph with six spokes.

Figure 9 below).

Finally, at the bottom, the sequence of atoms is shown as it was detected during recognition. For ease of understanding, the user can also click the *Merge* button to replace the first two atoms by their concatenation. This merging process is even animated. It is possible to keep clicking this button to iteratively merge the two leftmost graphs until the bottom contains only one graph, i.e., the concatenation of all the atoms at the beginning. The user can also save this visualization as a graphic file in SVG format.

4 Examples and Experiments

To confirm that graph recognition with finite automata can indeed run in linear time as claimed in Section 2 and [8], the recognition time was measured for graphs of four different graph languages. The graph languages are $a^n b^n c^n$ and *Spikes*, as introduced in Sections 2 and 3, respectively, and *Palindromes* and *Wheels*. *Palindromes* contains all graphs representing palindromes over $\{a,b\}$, and *Wheels* contains all wheel graphs as defined in [12, p. 92] and shown in Figure 10. The latter two are defined by

$$(a_{23}^{13}\, a_{31}^{32} \mid b_{23}^{13}\, b_{31}^{32})^*\, (a_{23}^{13}\, a_{\varepsilon}^{12} \mid b_{23}^{13}\, b_{\varepsilon}^{12} \mid a_{\varepsilon}^{12} \mid b_{\varepsilon}^{12}) \qquad \text{Palindromes}$$

$$t_{12}^{\varepsilon}\, s_{123}^{32}\, (t_{324}^{314}\, s_{123}^{123})^*\, t_{32}^{312}\, s_{\varepsilon}^{12} \qquad \text{Wheels}$$

The experiments consisted of running graph recognition using both the simple and the efficient edge selection algorithms (see Section 3.2) on input graphs of increasing size and measuring the execution time of the recognition algorithm. They were run on a 2022 Mac Studio with an Apple M1 Max processor, 64 GiB of RAM, and the Temurin 21 JAVA virtual machine. To get more accurate measurements, each recognition run was repeated 40 times on the same graph, shuffling the edges randomly each time to avoid any bias due to favorable or unfavorable ordering of the edges. The average execution time was then calculated by excluding the four worst times to exclude cases that were slowed down by garbage collection. The results of such executions are shown in Figures 11 and 12 for the languages $a^n b^n c^n$ and *Spikes*, respectively. In the plots, the x-axis shows the number of edges of the input graph and the y-axis shows the execution time in seconds. They clearly show that the runtime for the simple implementation of edge selection is quadratic and that the efficient implementation is indeed much faster.

In addition, Figure 13 compares the time taken to recognize graphs using only the efficient edge selection algorithm for the four languages on larger input graphs; in this case, we reduced the number of runs of each test from 40 to 6 in order to reduce the time required given the larger size of the graphs. It can be seen that the increase in execution time remains linear even when scaling up to large graphs with a million edges.

In addition to the recognition execution time, we also measured the time taken for the various operations required to create the DFA for the four languages. The operations of generating the automaton from a regular expression, checking the automaton for ambiguity, performing the powerset construction,

minimizing the DFA, and checking the DFA for the FEC property took less than 1 ms in total. Checking the TS property (and reordering the transitions) took on average 7.5 ms (between 6.8 ms and 8.3 ms). Since it is also possible to store the DFA resulting from these operations and reuse it for recognition operations, the time taken by these operations can be considered negligible.

Regarding the times shown in the graphs, in the case of the efficient edge selection implementation, they include both the time to generate the optimized data structures of the input graph and the time to execute the DFA. In particular, the time to create the optimized data structures was found to be about two-thirds of the total time, while the time to execute the DFA was about one-third of the total time.

5 Conclusions

This paper has described *Grappa RE*, a tool for analyzing graphs in the sense that it decides for a given graph whether it can be generated by a given finite automaton or not. *Grappa RE* implements the approach recently proposed by Hoffmann, Drewes, and Minas [8], where graphs are considered as interpretations of strings and finite automata are used to define a graph language and also as a device for deciding the membership problem. In particular, *Grappa RE* implements the procedures for checking whether a given automaton allows efficient graph recognition. This paper has shown that the recognition is indeed linear in the size of the input graph, and it has described the implementation approach that was necessary to achieve linear runtime complexity. Experiments have shown that even very large graphs can be analyzed very quickly with this implementation, since only about 2 μs is needed to analyze each edge of the input graph on standard computers.

Future work will consider applications of the approach described in [8], e.g., by using regular expressions to be used in the nested graph conditions of Habel and Pennemann [13, 18] to specify global graph properties and check them with automata.

References

[1] Michael A. Arbib & Yehoshafat Give'on (1968): *Algebra Automata I: Parallel Programming as a Prolegomena to the Categorical Approach*. *Information and Control* 12(4), pp. 331–345, doi:10.1016/S0019-9958(68)90374-4.

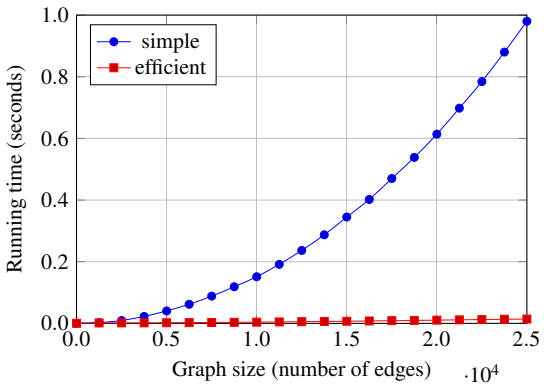


Figure 11: Recognizing $a^n b^n c^n$ graphs using the simple and the efficient implementation.

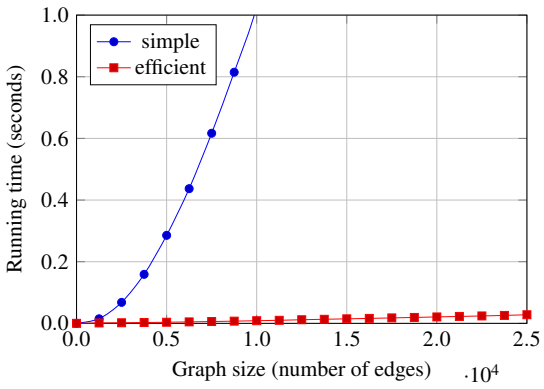


Figure 12: Recognizing *Spikes* graphs using the simple and the efficient implementation.

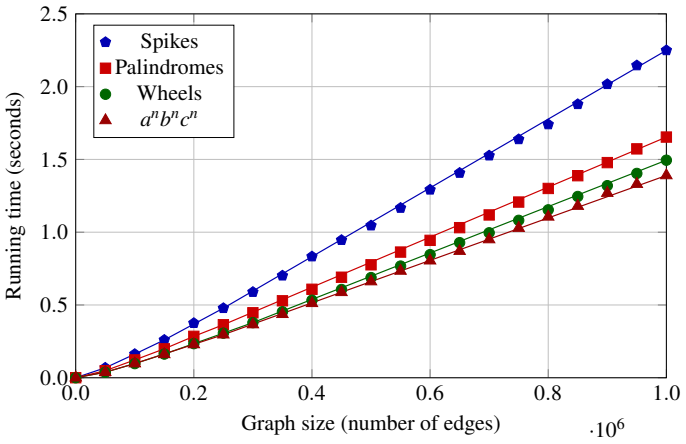


Figure 13: Recognizing different kinds of graphs using the efficient implementation.

[2] Stefan Balev, Antoine Dutot, Yoann Pigné, Guilhelm Savin et al. (2024): *GraphStream - A Dynamic Graph Library*. Available at https://graphstream-project.org/.

[3] Christoph Blume, H.J. Sander Bruggink, Martin Friedrich & Barbara König (2013): *Treewidth, Pathwidth and Cospan Decompositions with Applications to Graph-Accepting Tree Automata*. *Journal of Visual Languages & Computing* 24(3), pp. 192–206, doi:10.1016/j.jvlc.2012.10.002.

[4] Symeon Bozapalidis & Antonios Kalampakas (2008): *Graph automata*. *Theoretical Computer Science* 393(1-3), pp. 147–165, doi:10.1016/j.tcs.2007.11.022.

[5] Frank Drewes, Berthold Hoffmann & Mark Minas (2015): *Predictive Top-Down Parsing for Hyperedge Replacement Grammars*. In Francesco Parisi-Presicce & Bernhard Westfechtel, editors: *Graph Transformation - 8th International Conf., ICGT 2015. Proceedings*, *Lecture Notes in Computer Science* 9151, Springer, pp. 19–34, doi:10.1007/978-3-319-21145-9_2.

[6] Frank Drewes, Berthold Hoffmann & Mark Minas (2019): *Formalization and Correctness of Predictive Shift-Reduce Parsers for Graph Grammars based on Hyperedge Replacement*. *Journal on Logical and Algebraic Methods for Programming* 104, pp. 303–341, doi:10.1016/j.jlamp.2018.12.006.

[7] Frank Drewes, Berthold Hoffmann & Mark Minas (2021): *Rule-Based Top-Down Parsing for Acyclic Contextual Hyperedge Replacement Grammars*. In Fabio Gadducci & Timo Kehrer, editors: *Graph Transformation - 14th International Conference, ICGT 2021*, *Lecture Notes in Computer Science* 12741, Springer, pp. 164–184, doi:10.1007/978-3-030-78946-6_9.

[8] Frank Drewes, Berthold Hoffmann & Mark Minas (2024): *Finite Automata for Efficient Graph Recognition*. In: *Proceedings 14th International Workshop on Graph Computation Models, Leicester, UK, 18th July 2023*, *Electronic Proceedings in Theoretical Computer Science* 417, Open Publishing Association, pp. 134–156, doi:10.4204/EPTCS.417.8.

[9] Matthew Earnshaw & Paweł Sobociński (2022): *Regular Monoidal Languages*. In Stefan Szeider, Robert Ganian & Alexandra Silva, editors: *47th International Symposium on Mathematical Foundations of Computer Science (MFCS 2022)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 241, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 44:1–44:14, doi:10.4230/LIPIcs.MFCS.2022.44.

[10] Joost Engelfriet & Jan Joris Vereijken (1997): *Context-Free Graph Grammars and Concatenation of Graphs*. *Acta Informatica* 34(10), pp. 773–803, doi:10.1007/s002360050106.

[11] Yehoshafat Give'on & Michael A. Arbib (1968): *Algebra Automata II: The Categorical Framework for Dynamic Analysis*. *Information and Control* 12(4), pp. 346–370, doi:10.1016/S0019-9958(68)90381-1.

[12] Annegret Habel (1992): *Hyperedge Replacement: Grammars and Languages*. *Lecture Notes in Computer Science* 643, Springer, doi:10.1007/BFb0013875.

[13] Annegret Habel & Karl-Heinz Pennemann (2005): *Nested Constraints and Application Conditions for High-Level Structures*. In H.-J. Kreowski et al., editors: *Formal Methods in Software and System Modeling*, *Lecture Notes in Computer Science* 3393, Springer, pp. 293–308, doi:10.1007/978-3-540-31847-7_17.

[14] Berthold Hoffmann & Mark Minas (2017): *Generating Efficient Predictive Shift-Reduce Parsers for Hyperedge Replacement Grammars*. In M. Seidl & S. Zschaler, editors: *STAF 2017 Workshops*, *Lecture Notes in Computer Science* 10748, Springer, pp. 76–91, doi:10.1007/978-3-319-74730-9_7.

[15] John Hopcroft (1971): *An $n \log n$ Algorithm For Minimizing States In A Finite Automaton*. In Zvi Kohavi & Azaria Paz, editors: *Theory of Machines and Computations*, Academic Press, pp. 189–196, doi:10.1016/B978-0-12-417750-5.50022-1.

[16] Antonios Kalampakas (2011): *Graph Automata: The Algebraic Properties of Abelian Relational Graphoids*. In Werner Kuich & George Rahonis, editors: *Algebraic Foundations in Computer Science - Essays Dedicated to Symeon Bozapalidis on the Occasion of His Retirement*, *Lecture Notes in Computer Science* 7020, Springer, pp. 168–182, doi:10.1007/978-3-642-24897-9_8.

[17] Anders Møller (2021): *dk.brics.automaton – Finite-State Automata and Regular Expressions for Java*. Available at http://www.brics.dk/automaton/.

[18] Karl-Heinz Pennemann (2009): *Development of Correct Graph Transformation Systems*. Dissertation, Carl-von-Ossietzky-Universität Oldenburg. Available at http://oops.uni-oldenburg.de/884/1/pendev09.pdf.

Scalable Pattern Matching in Computation Graphs

Luca Mondada
University of Oxford
Oxford, UK
Quantinuum Ltd
Cambridge, UK
luca.mondada@cs.ox.ac.uk

Pablo Andrés-Martínez
Quantinuum Ltd
Cambridge, UK

Graph rewriting is a popular tool for the optimisation and modification of graph expressions in domains such as compilers, machine learning and quantum computing. The underlying data structures are often port graphs—graphs with labels at edge endpoints. A pre-requisite for graph rewriting is the ability to find graph patterns. We propose a new solution to pattern matching in port graphs. Its novelty lies in the use of a pre-computed data structure that makes the pattern matching runtime complexity independent of the number of patterns. This offers a significant advantage over existing solutions for use cases with large sets of small patterns.

Our approach is particularly well-suited for quantum superoptimisation. We provide an implementation and benchmarks showing that our algorithm offers a 20x speedup over current implementations on a dataset of 10 000 real world patterns describing quantum circuits.

1 Introduction

Optimisation of computation graphs is a long-standing problem in computer science that is seeing renewed interest in the compiler [12], machine learning (ML) [8, 5] and quantum computing communities [20, 19]. In all of these domains, graphs encode computations that are either expensive to execute or that are evaluated repeatedly over many iterations, making graph optimisation a primary concern.

Domain-specific heuristics are the most common approach in compiler optimisations [14, 17]— a more flexible alternative are optimisation engines based on *rewrite rules*, describing the allowable graph transformations [3, 4]. Given a computation graph as input, we find a sequence of rewrite rules that transform the input into a computation graph with minimal cost. One successful approach in both ML and quantum computing has been to use automatically generated rules, scaling to using hundreds and even thousands of rules [20, 8, 19].

In the implementations cited above, pattern matching is carried out separately for each pattern, becoming a bottleneck for large rule sets. We present an algorithm for pattern matching on computation graphs that uses a pre-computed data structure to return all pattern matches in a single query. The set of rewrite rules are directly encoded in this data structure: after a one-time cost for construction, pattern matching queries can be answered in running time independent of the number of rewrite rules.

We provide a novel solution to pattern matching on port graphs [6] with a runtime complexity independent of the number of patterns. As a trade-off, the runtime may be exponential in the size of the patterns. For pattern sizes of practical interest in quantum computing, however, the resulting costs are manageable: the exponential scaling is in the number of qubits of the patterns, which is bounded by a single digit constant in relevant rewriting use cases [20].

The solution we propose can be seen as an adaptation of Rete networks [7] to the special case of port graph pattern matching. The additional structure obtained from restricting our considerations to graphs results in a simplified network design and crucially, allows us to derive worst-case asymptotic

J. Endrullis, D. Grzelak, T. Heindel and J. Kosiol (Eds):
GCM 2023 and GCM 2024
EPTCS 417, 2025, pp. 71–95, doi:10.4204/EPTCS.417.5

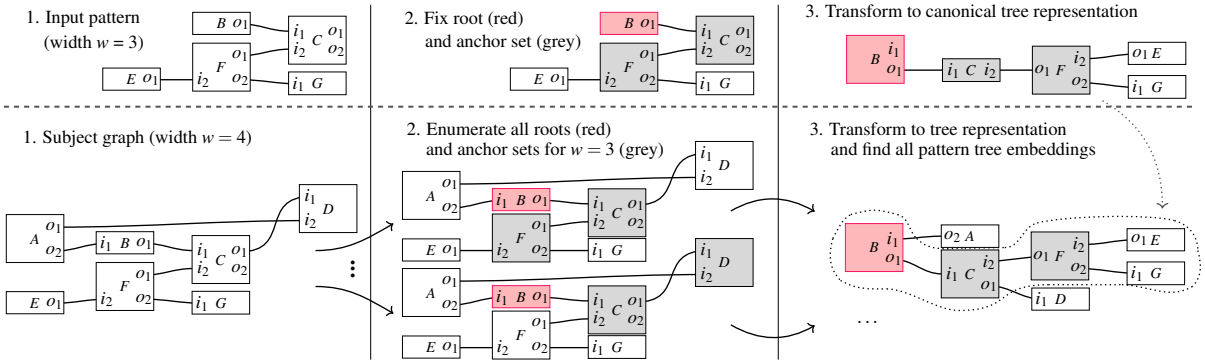


Figure 1: Pattern matching on a port graph is reduced to the problem of matching on trees. A subset of vertices are chosen as anchor sets (left). A neighbourhood of the anchors is extracted and represented as a tree (middle). Finally, pattern matches are found by searching for matching subtrees (right).

runtime bounds—overcoming a key limitation of Rete. A similar problem is also studied in the context of multiple-query optimisation for database queries [16, 15], but has limited itself to developing caching strategies and search heuristics for specific use cases. Finally, using a pre-compiled data structure for pattern matching was already proposed in [13]. However, with a $n^{\Theta(m)}$ space complexity—n is the input size and m the pattern size—it is a poor candidate for pattern matching on large graphs, even for small patterns.

2 Paper overview

Taking advantage of *port labels* on graph data structures leads to a speedup for pattern matching over the general case [10]. Port labels are data assigned to every endpoint of the edges of a graph, such that the labels at every vertex are unique. Such labels can for instance be assigned to processes with distinguishable inputs and outputs: a function that maps inputs $(x_1,\ldots,x_n)$ to output $(y_1,\ldots,y_m)$ can assign labels i_1 to i_n and o_1 to o_m to its incident edges in the computation graph. The resulting data structure is a *port graph* [6].

Main idea. For a set of ℓ pattern port graphs $P_1,\ldots,P_\ell$ and a subject port graph G, we solve the problem of finding all pattern embeddings $P_i \to G$. We distinguish two separate stages during pattern matching:

> *Pre-computation stage.* Compile the set of patterns into a data structure designed to speed up later queries. In the process, we select for every pattern P_i an *anchor set*, i.e. a subset X_i of vertices in P_i. The input port graph G is not required at this stage and, hence, this computation is done only once for a given collection of patterns $P_1,\ldots,P_\ell$.
>
> *Fast pattern matching stage.* Given G, compute for each P_i all embeddings $P_i \to G$. This is achieved by enumerating all possible images X in G of pattern anchor sets $X_1,\ldots,X_\ell$ and finding the subset of patterns i for which the map $X_i \to X$ can be extended to a valid pattern embedding of P_i in G.

The pattern matching stage can be decomposed into known problems by showing that embeddings with fixed anchor sets can be equivalently seen as rooted tree embeddings. The enumeration of valid choices of X in G then becomes a tree counting argument. Meanwhile, the set of patterns that embed in G for a fixed $X_i \to X$ is obtained from a pre-computed decision tree. An overview is presented in fig. 1.

Results and contributions. Our first major contribution is a pattern matching algorithm for port graphs with a runtime complexity bound independent of the number of patterns being matched, achieved using a one-off pre-computation. The main complexity result is expressed in terms of maximal pattern *width* w and *depth* d, two measures of pattern size defined in section 3.1. These are directly related to the tree representation illustrated in fig. 1: width is equal to the size of the anchor set (proposition 4) and depth is at most twice the tree height (eq. (7)). We assume bounded degree graphs (the complete list of assumptions is given in section 3.2) and we use the *graph size* $|G|$ to refer to the number of vertices in G.

Theorem 1. *Let $P_1, \dots, P_\ell$ be patterns with at most width w and depth d. The pre-computation runs in time and space complexity*

$$O\left((d \cdot \ell)^w \cdot \ell + \ell \cdot w^2 \cdot d\right).$$

For any subject port graph G, the pre-computed prefix tree can be used to find all pattern embeddings $P_i \to G$ in time

$$O\left(|G| \cdot \frac{c^w}{w^{1/2}} \cdot d\right) \tag{1}$$

where $c = 6.75$ is a constant.

The runtime complexity is dominated by an exponential scaling in maximal pattern width w. Meanwhile, the advantage of our approach over matching one pattern at a time grows with the number of patterns ℓ. It is thus of particular interest for matching numerous small width patterns.

We illustrate this point by comparing our approach to a standard algorithm that matches one pattern at a time [10], with runtime complexity $O(\ell \cdot |P| \cdot |G|)$. Using $|P| \leq w \cdot d$ (shown in section 3.1) and comparing to eq. (1), we thus have a speedup in the regime $\Theta(c^w/w^{3/2}) < \ell$. On the other hand, ℓ is upper bounded by the maximum number $N_{w,d}$ of patterns of bounded width and depth. Using a crude lower-bound for $N_{w,d}$ derived in appendix B, we obtain a computational advantage for our approach when

$$\Theta\left(\frac{c^w}{w^{3/2}}\right) < \ell < \left(\frac{w}{2e}\right)^{\Theta(wd)} \leq N_{w,d}. \tag{2}$$

Our second major contribution is an efficient Rust library for port graph pattern matching[1]. We present benchmarks on a real world dataset of 10 000 quantum circuits in section 5, showing a 20× speedup over a leading C++ implementation of pattern matching for quantum circuits.

3 Preliminaries

3.1 Definitions

A port graph G is a tuple $G := (V, E, \mathscr{P}, \lambda)$ where (V, E) is an undirected multigraph, $\mathscr{P}$ is a finite set of *port labels* and $\lambda : V \times \mathscr{P} \rightharpoonup E \cup \{\omega\}$ is a partial function that on its domain of definition either assigns port labels to edges or marks them as open ports using a specially dedicated symbol ω. The port graph is valid if $\lambda(v,p) = \lambda(v,p') = e \in E$ if and only if e is an edge incident in v and $p = p'$. We then say that e is attached to v at port p. The domain of definition of $\lambda(v,\cdot)$ is the set of port labels at vertex v, written $ports(v)$. The degree of v is $deg(v) = |ports(v)|$. It will often be convenient to leave the definition of λ implicit and for an edge $e = \lambda(v,p) = \lambda(v',p')$, to write it instead as the set $e = \{(v,p),(v',p')\}$. Additionally, we may consider port graphs with labelled vertices, determined by *vertex label maps* $V \to \mathscr{W}$, where $\mathscr{W}$ is a set of labels.

[1] portmatching: `https://github.com/lmondada/portmatching`

Width, depth and linear paths. Fix a partition of port labels $\mathscr{P}$ into pairs of elements (and an additional singleton set if $|\mathscr{P}|$ is odd). This defines an equivalence relation $\sim$ where $p \sim p'$ if p and p' are in the same partition. The relation $\sim$ defines a partition of the edges of a port graph into paths. A linear path in a port graph G with edges E and port labels in $\mathscr{P}$ is a path $P \subseteq E^*$ such that for every vertex v in G and ports $p, p' \in \mathscr{P}$ satisfying $p \sim p'$,

$$\lambda(v,p) \in P \quad \Longrightarrow \quad \lambda(v,p') \in P \cup \{\omega\}. \tag{3}$$

From a single edge in G, a linear path can be constructed uniquely by repeatedly appending edges to the path so that (3) is satisfied. The linear path decomposition of G is the unique partition of the edges of G into linear paths. The width $width(G)$ of G is the number of linear paths in this decomposition. The depth $depth(G)$ of G is the length of the longest linear path in G. Every edge is on exactly one linear path, while vertices may be on zero, one or several linear paths. We have always $|G| \leq width(G) \cdot depth(G)$.

Patterns and embeddings. A pattern is a connected port graph. A pattern embedding (or just embedding) $\varphi : P \to G$ from a pattern $P = (V_P, E_P, \mathscr{P}, \lambda_P)$ to a subject port graph $G = (V, E, \mathscr{P}, \lambda)$, both with identical port label sets, is given by an injective vertex map $\varphi_V : V_P \to V$ such that $\lambda_P(v,p)$ is defined if and only if $\lambda(\varphi_V(v), p)$ is defined and the edge map $\varphi_E : E_P \to E$ defined as

$$\varphi_E(e) = \lambda(\varphi_V(v), p) \quad \text{for } (v,p) \in V \times \mathscr{P} \text{ s.t. } \lambda_P(v,p) = e \tag{4}$$

is well-defined and injective. Finally, if the pattern and port graphs have node labels $V_P \to \mathscr{W}$ and $V \to \mathscr{W}$, then we also require that pattern embeddings preserve those.

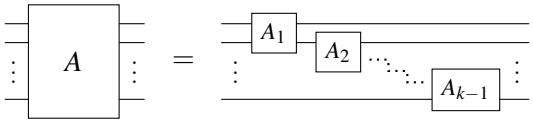
3.2 Simplifying assumptions

We list here a series of assumptions made throughout our argument. They represent a restriction from the most general case but we do not find that they restrict the usefulness of the result in practice. We show in section 3.3 that these assumptions hold in the case of quantum circuits. Moreover, as discussed in section 5, none of these assumptions are required for the implementation, and we have not observed any impact on performance when lifting them in practice, so we conjecture that these assumptions can be loosened and our results generalised.

1. All graphs and patterns are of bounded maximum degree Δ.
2. There is no linear path that forms a cycle.
3. All pattern embeddings $\varphi : P \to G$ must be *convex*, i.e. for every subgraph $H \subseteq G$ that contains the image of P, $\varphi(P) \subseteq H$, it holds that $width(P) \leq width(H)$.

Note also that eq. (4) requires that the degree of a vertex v in P is also preserved $deg(v) = deg(\varphi(v))$. Importantly, a pattern embedding may map a vertex with open port p to a vertex in the subject graph that has an edge attached to port p.

We will further simplify the problem by making choices of presentation that do not imply any loss of generality. First of all, we will assume that all vertices are on at most two linear paths (and thus in particular $\Delta = 4$). Vertices on $k > 2$ linear paths can always be broken up into a composition of $k-1$ vertices, each on two linear paths as follows:



This transformation leaves graph width unchanged but may multiply the graph depth by up to a factor Δ. We can then fix the set of port labels to the set $\mathscr{P} = \{i_1, i_2, o_1, o_2\}$ with a total order $\leq$ on $\mathscr{P}$[2]. We fix the partition of $\mathscr{P}$ into pairs of elements given by $i_k \sim o_k$ for $k \in \{1,2\}$. We also enforce that at every vertex v, the set of ports $ports(v)$ is partitioned by $\sim$ into $\lfloor ports(v)/2 \rfloor$ pairs of elements and at most one singleton set. This can always be achieved by relabelling vertex ports. Finally, we assume that all patterns have the same width w and depth d, are connected port graphs and have at least 2 vertices.

Using these assumptions we can obtain the following notable bound on graph width.

Proposition 2. *Let G be a port graph with n_{odd} vertices of odd degree and n_ω open ports. Then the graph width of G is at most $\lfloor (n_{odd} + n_\omega)/2 \rfloor$.*

The proof is in the appendix A.1.

Rooted paths ordering. The total order on $\mathscr{P}$ also induces a total order on the paths $e_1 \cdots e_k \in E^*$ in G that start in the same vertex $r \in e_1$: the paths are equivalently described by a string in $\mathscr{P}^*$, the sequence of ports of $e_1, \ldots, e_k$, which we order using the lexicographical ordering on strings. For every vertex v in G there is thus a unique smallest path from r to v in G that is invariant under isomoprhism of the underlying graph (i.e. relabelling of the vertex set).

3.3 Quantum Circuits

We see pattern matching for quantum circuits as one of the main applications of our results. We therefore choose to introduce here the quantum circuit syntax as a motivation and illustration of the port graph formalism. Similar data structures are also in use in other parts of compilation science, variously referred to as circuits, computation graphs or dataflow graphs. Readers familiar with port graphs or uninterested in the application in quantum computing may skip directly to the definitions of section 3.1.

The set of operations in a quantum circuit is called the *gate set* of the computation and forms the set of node weights of the port graph. To every element of the gate set, called a *gate type*, is associated a gate arity. A gate with gate type of arity n has n incoming port labels i_k and n outgoing port labels o_k—by our assumptions we thus assume $1 \leq n \leq 2$. Edges always connect outgoing to incoming labels, and the directed port graph that results from these edge orientations is acyclic. A quantum circuit has q qubits if it has q outgoing and q incoming open ports: the inputs and outputs of the circuit.

All assumptions made in section 3.2 can be easily verified for quantum circuits and in fact will also apply to most computation graphs more generally. Indeed:

1. Bounded degree is a direct consequence of a having a fixed set of gate types.
2. For any directed acyclic computation, using port labels in i_k for incoming ports and o_k for outgoing labels will always result in non-cyclic linear paths.
3. Similarly, convexity is a natural restriction in the context of rewriting systems for acyclic digraphs, as it ensures that the application of a rewrite rule does not introduce a cycle in the graph.

Using the $i_k \sim o_k$ port label partition, linear paths in a quantum circuit correspond to the paths of gates along a single qubit. For applications to quantum circuits, we can thus bound graph width and depth as follows:

Proposition 3. *The port graph G of a quantum circuit with q qubit and at most d gates on any one qubit has width q and depth d.* □

Some further considerations on applying our work to quantum circuits are discussed in appendix C.

[2] Any total order will work, e.g. $i_1 \leq i_2 \leq o_1 \leq o_2$.

3.4 Pattern Matching

In our pattern matching task, given a subject port graph G and a collection of port graphs $P_1, \ldots, P_\ell$, we must find all pattern embeddings

$$\{\varphi : P_i \to G \mid 1 \leq i \leq \ell \text{ s.t. } \varphi \text{ is a pattern embedding}\}. \tag{5}$$

Finding pattern embeddings in port graphs is a simple problem already studied in other contexts [9, 10]. As a result of eq. (4), for every vertex r in P and r_G in G there can be at most one embedding $P \to G$ that maps r to r_G: for $v \neq r$ in P, there is a path in P from r to v which, viewed as a sequence of port labels, maps uniquely to a path in G starting at r_G and ending in the image of v. For a choice of r in P it is therefore sufficient to consider every posible image r_G in G to find all embeddings $P \to G$.

We are however interested in the regime where the number of patterns ℓ may be large and pattern matching is performed many times for the same set of patterns. For this scenario it makes sense to proceed in two steps and introduce *pattern matching with pre-computation*. Given patterns $P_1, \ldots P_\ell$, we first produce a *pattern matcher*, a program whose representation can be stored to disk. In a second step, a subject port graph G is passed to the pattern matcher, which computes the set (5). We are interested in two properties of the solution:

- What is the complexity of pattern matching on input G given such a pattern matcher?
- What is the time complexity of generating a pattern matcher given patterns, and what is the size of the pattern matcher that is produced?

The answer to the first question is our main concern: for a fixed collection of patterns, this will determine the runtime to obtain all pattern embeddings given an input diagram. The second question, on the other hand, primarily concerns a one-off pre-computation step that only needs to be performed once for any set of patterns. In practice, this may also impinge on the first question, if the matcher does not fit into RAM and/or CPU cache.

4 Algorithm description

4.1 Canonical Tree Representation

Connected port graphs admit an equivalent representation as trees, which we will use for matching.

Split Graphs. Let G be a connected port graph with vertices V and consider the linear path decomposition of G. In this decomposition every vertex v in G must be on one or more linear paths. We mark a subset $X \subseteq V$ of vertices of G as 'immutable' and split every other vertex $v \in V \setminus X$ into multiple vertices, rewiring the edges in such a way that all vertices not in X are now on exactly one linear path. We call the graph thus obtained the X-split graph of G, and write it $split_X(G)$. Figure 2 shows an example of a graph and its split graph. Formally, the split graph can be characterised using an equivalence relation $\equiv$ given by

$$(v, p) \equiv (v', p') \quad \Leftrightarrow \quad v = v' \wedge (v \in X \vee p \sim p') \tag{6}$$

The vertices of the split graph are the equivalence classes of $\equiv$ and the edges are obtained by mapping the edges of G one to one: two vertices in the X-split graph are connected by an edge if and only if there is an edge in G between some elements of their equivalence classes. Note that if the anchor set X is too small, $split_X(G)$ may not be connected; e.g. if $X = \varnothing$ there would be $width(G)$ connected components, one per linear path.

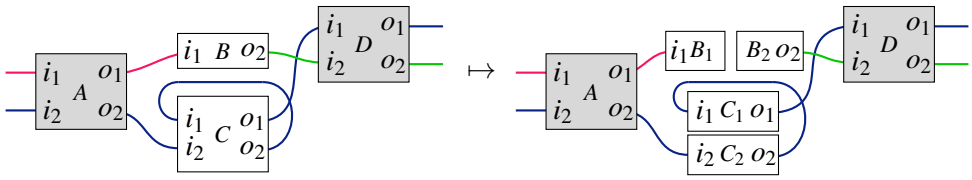


Figure 2: A port graph and its linear path decomposition (coloured edges) on the left. On the right, the split graph resulting from the choice of anchors $X = \{A, D\}$. We use the circuits convention, i.e. port labels are partitioned into linear paths using the relation $i_k \sim o_k$.

A recovery of the original port graph G given $\text{split}_X(G)$ is made possible by adding vertex labels to the split graph that identify split vertices. In the following, split graphs are always rooted trees, i.e. connected acyclic graphs with a chosen root vertex. Using paths of port labels we obtain a vertex labelling that is invariant under pattern embedding. This is discussed in more details in the context of the CT representation in appendix D.

Anchor sets. If $split_X(G)$ is connected and acyclic, we say that X is an anchor set of G and call the vertices in X anchor vertices. If $width(G) > 1$, then every linear path in G must contain at least one anchor vertex. A set of $width(G)$ anchors always exists and can be computed constructively:

Proposition 4. *For a connected port graph G of width w and depth d and for a vertex r of G, listing 1 returns an anchor set of w vertices; we call this the set of canonical anchors. Its runtime is $O(w^2 \cdot d)$.*

The proof is in appendix A.2. Note that the code given assumes that the linear paths of G are already computed. These can be computed at the beginning of the computation in time linear in the graph size—and thus do not affect the overall asymptotic complexity. As a direct consequence we have the following:

Corollary 5. *For a port graph G and root r_G in G: $width(G) = |\textsc{CanonicalAnchors}(G, r_G)|$.* □

CT representation. We call the tree $\text{split}_X(G)$ obtained from the canonical anchors, along with the choice of root r, the **canonical tree** (CT) representation of G. For simplicity, we will assume that the root is chosen such that it is on a single linear path[3]. Non-root internal nodes with more than one child are contained in the set of anchor vertices of G, every leaf is at most a height $depth(G)$ below the nearest anchor and there is at least one leaf a distance $depth(G)$ from the nearest anchor. As a consequence the CT tree width t_w and height t_h can be bound by

$$t_w \leq (\Delta - 2) \cdot width(G) = 2 \cdot width(G) \quad \text{and} \quad depth(G)/2 \leq t_h \leq width(G) \cdot depth(G), \tag{7}$$

where $\Delta = 4$ is the maximum degree of G. Using CT representations of patterns simplifies the pattern matching problem for three reasons:

- Connected port graphs are uniquely characterised by their CT representation, i.e. there is an injective map from patterns with a choice of root to their CT representation.
- Every vertex in the CT representation is either the root vertex or it is uniquely identified by the path to it from the root. Paths can be defined by port labels, which are invariant under pattern embeddings.
- Rooted trees are uniquely characterised by a partition of their edges into paths, which can in turn be encoded as strings. See fig. 3 for an example.

The properties of the CT representation is discussed in more details in appendix D.

[3] We can ensure that such a root always exists for example by adding dummy vertices on every edge.

Listing 1: Finding the set of canonical anchors. CANONICALANCHORS is a convenience wrapper around CONSUMEPATH, which is defined recursively. The latter returns not only the anchor list, but also the updated set of seen linear paths. `G.linear_paths(v)` is the set of all linear paths in G that go through vertex v. For traversal, a linear path `lp` is split into two paths starting at vertex v using `lp.split_at(v)`. The sequence of vertices starting from v to the end of the path is represented as a queue. The symbol ++ designates list concatenation.

```
function CanonicalAnchors(G: Graph, root: Vertex) -> List[Vertex]:
  # Initialise the variables for ConsumePath and return the anchors
  (anchors, seen_paths) = ConsumePath(G, [root], ∅):
  return anchors

function ConsumePath(
    G: Graph,
    path: Queue[Vertex],
    seen_paths: Set[LinearPath],
) → (List[Vertex], Set[LinearPath]):
  new_anchor = null
  unseen = ∅
  # Find the first vertex in the queue on an unseen linear path
  while unseen == ∅:
    if path.is_empty():
      return ([], {})
    new_anchor = path.pop()
    unseen = G.linear_paths(new_anchor) \ seen_paths

  # Add the new linear paths to the set of seen paths
  seen_paths = seen_paths ∪ unseen

  # We traverse the rest of current path as well as all the new linear paths
  paths = [path]
  for lp in unseen:
    (left_path, right_path) = lp.split_at(new_anchor)
    paths.push(left_path)
    paths.push(right_path)

  # For each path find anchors recursively and update seen paths
  anchors = [new_anchor]
  for path in paths:
    (new_anchors, new_seen_paths) = ConsumePath(G, path, seen_paths)
    anchors = anchors ++ new_anchors
    seen_paths = new_seen_paths
  return (anchors, seen_paths)
```

4.2 Pattern matching with fixed anchors

We aim to present an ℓ-independent pattern matcher: an algorithm that can identify all pattern embeddings in a subject graph (5) whose complexity is independent from the number of patterns ℓ. In this section, we restrict to pattern embeddings that map the set of canonical anchors of the pattern to a fixed subset of vertices in the subject graph, and we show that this problem can be reduced to a simple matching problem on strings.

We start by observing that such pattern embeddings with fixed anchors correspond to tree inclusions of CT representations. Let G be a port graph, let $P_1, \dots, P_\ell$ be patterns of width w and let $X \subseteq V$ be a set of w vertices in G. Choose root vertices $r_G \in X$ and r_i in patterns P_i. Write T_i for the CT representation of P_i rooted in r_i. Consider the following set $\mathscr{G}$ of subgraphs of G:[4]

$$\mathscr{G} = \{H \subseteq G \mid H \text{ is connected convex subgraph and } \textsc{CanonicalAnchors}(H, r_G) = X\}. \quad (8)$$

Proposition 6. *If $\mathscr{G} \neq \varnothing$, then there is a connected subgraph $G_{max} \subseteq G$ such that $H \subseteq G_{max}$ for all $H \in \mathscr{G}$. The split graph $split_X(G_{max})$ is a tree rooted in r_G. There is a pattern embedding $\varphi\colon P_i \to G$ mapping the canonical anchor set of P_i to X and $\varphi(r_i) = r_G$ if and only if there is an injective embedding of trees $\phi\colon T_i \to split_X(G_{max})$ with $\phi(r_i) = r_G$ that satisfies eq.* (4) *and preserves vertex labels.*

The proof gives an explicit construction for $G_{\max}$.

Proof. Let $L_1, \dots, L_w$ be the subset of linear paths in G that go through at least one vertex in X. Let $G_{\max} \subseteq G$ be the subgraph of G defined by the edges

$$E_{\max} = \bigcup_{1 \leq i \leq w} L_i, \quad (9)$$

For any subgraph $H \in \mathscr{G}$ it holds that $H \subseteq G_{\max}$ due to the linear paths of H being contained in $L_1, \dots, L_w$. Since any $H \in \mathscr{G}$ is connected, the anchors in X are connected in H and therefore also in $G_{\max}$. As a consequence, all vertices of $G_{\max}$ are connected. The port graph $\text{split}_X(G_{\max})$ must be a tree, as otherwise its canonical anchors would be a strict subset of X and by corollary 5, $width(\text{split}_X(G_{\max})) < |X|$. Hence,

$$width(G_{\max}) = width(\text{split}_X(G_{\max})) < |X| = width(H).$$

contradicting the convexity assumption of eq. (8). Assuming $\mathscr{G} \neq \varnothing$, we now prove the bidirectional implication between φ and ϕ.

$\Leftarrow$: We use vertex labels on T_i and $\text{split}_X(G_{\max})$ to mark with a unique label vertices that were split from a same vertex v in P_i, respectively $G_{\max}$ (details in appendix D). Let $\mathscr{V}_{P_i}$ and $\mathscr{V}_{G_{\max}}$ be the partition of the vertices of T_i and $\text{split}_X(G_{\max})$ into sets of split vertices with identical labels; there are bijective maps between $\mathscr{V}_{P_i}$ and the vertices in P_i as well as between $\mathscr{V}_{G_{\max}}$ and the vertices in $G_{\max}$. The tree embedding $\phi\colon T_i \to \text{split}_X(G_{\max})$ preserves vertex labels and thus maps sets in $\mathscr{V}_{P_i}$ to sets in $\mathscr{V}_{G_{\max}}$: it thus defines a map $\varphi_V : P_i \to G_{\max}$.

φ_V is injective by injectivity of ϕ and maps the root r_i to r_G by construction. The $w-1$ non-root anchor vertices of T_i are the only vertices in T_i on more than one linear path: they must be mapped to the $w-1$ vertices in $\text{split}_X(G_{\max})$ with the same properties—precisely its non-root anchor vertices. Edges are mapped bijectively by graph splitting and thus the map φ_E can be defined by using ϕ_E and must satisfy eq. (4). Since $G_{\max} \subseteq G$, we conclude that φ is a valid pattern embedding $P_i \to G$.

[4] A convex subgraph $H \subseteq G$ is one such the canonical embedding $H \to G$ is convex, as defined in assumption 3 of section 3.2.

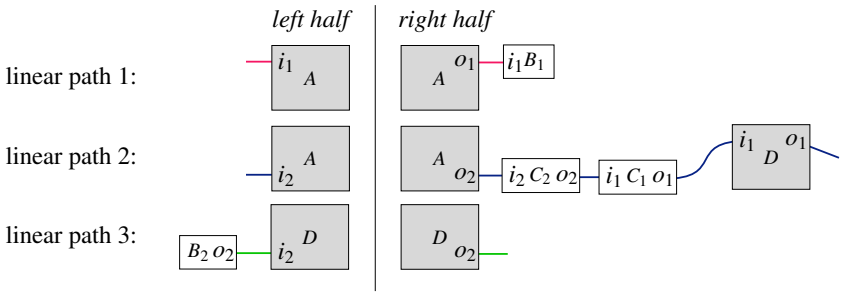


Figure 3: The split graph of fig. 2, represented by 6 strings obtained from its 3 linear paths.

$\Rightarrow$: The image of $\varphi : P_i \to G$ is a convex connected subgraph of G with canonical anchors X and root r_G, and thus is in $\mathscr{G}$. It must in particular be a subgraph of $G_{\max}$, and thus we can view φ as an embedding $\varphi : P_i \to G_{\max}$.

Note that edges are mapped bijectively between split and unsplit graphs, and thus the pattern embedding φ defines an injective map ϕ_E from edges in T_i to edges in $\text{split}_X(G_{\max})$. We construct the map $\phi : T_i \to \text{split}_X(G_{\max})$ inductively over the vertex set of T_i. We start by defining $\phi(r) := r_G$. Using eq. (4) and ϕ_E, we can then uniquely define the image of any neighbouring vertex of r in T_i. Proceeding inductively we will define ϕ on all vertices of T_i since it is connected. Because eq. (4) holds on φ, this procedure is well-defined and the resulting map ϕ will also satisfy eq. (4).

Now suppose v, v' are vertices in T_i such that $\phi(v) = \phi(v')$. By the inductive construction there are paths from the root r to v and v' respectively such that their image under ϕ_E are two paths from r_G to $\phi(v) = \phi(v')$. But $\text{split}_X(G_{\max})$ is a tree, so both paths must be equal. By bijectivity of ϕ_E, it follows $v = v'$, and thus ϕ is injective. Finally, the vertex labels are defined to be invariant under pattern embedding and thus are preserved by definition. □

Given G and a vertex set X we can thus find a maximal subgraph $G_{\max} \subseteq G$ that contains all subgraphs of G with canonical anchors X. Given $P_1, \ldots, P_\ell$, X and G, we then compute the CT representations of $P_1, \ldots, P_\ell$ and check for inclusions within $\text{split}_X(G_{\max})$. We will use an ℓ-independent tree matching algorithm for the latter task, thus solving the ℓ-independent pattern matching problem on port graphs.

String encoding of CT representations. We reduce the tree inclusion problem that results from proposition 6 to a string prefix matching problem that admits a well-known solution, discussed in proposition 14 of appendix E. The main idea is to partition the edges of the CT representation into linear paths, each of which is represented by two strings. They encode the paths from the anchor vertex on the path to either end of the linear path by expressing them as sequences of port labels. A graph of width w will have w linear paths and will be split into $2w$ strings. For the example graph of fig. 2, we obtain six split linear paths, shown in fig. 3. See appendix D for more details.

This encoding defines the $\text{AsStrings}(T, r)$ procedure: it takes as input a connected acyclic split graph T and a root r in T, and returns an encoding of T and its vertex labels as $2 \cdot \mathit{width}(T)$ strings.

Proposition 7. *Let T_1, T_2 be acyclic connected split graphs of width w and let r_1, r_2 be vertices in T_1 resp. T_2. Let $(s_1, \ldots, s_{2w}) = \text{AsStrings}(T_1, r_1)$ and $(t_1, \ldots, t_{2w}) = \text{AsStrings}(T_2, r_2)$ be the string encodings of their linear paths. Then there is an injective tree embedding $T_1 \to T_2$ that satisfies eq. (4), maps r_1 to r_2 and preserves vertex labels if and only if $s_i \subseteq t_i$ for all $1 \leq i \leq 2w$.*

The proof consists in showing that trees can be fully defined by the set of all paths from the root, which can be encoded in and recovered from the string representation. The proof is in appendix A.3.

The $\subseteq$ notations on string designates prefix inclusion. The string prefix matching problem is a simple computational task that can be generalised to to check for multiple string patterns at the same time. An overview of this problem can be found in appendix E. Putting propositions 6 and 7 together, we can thus obtain a solution for the ℓ-independent pattern matching problem for fixed anchors:

Proposition 8. *Let G be a port graph, $P_1, \dots, P_\ell$ be patterns of width w and depth at most d, and $X \subseteq V$ be a set of w vertices in G. The set of all pattern embeddings mapping the canonical anchor set of P_i to X and root r_i to r_G for $1 \leq i \leq \ell$ can be computed in time $O(w \cdot d)$ using a pre-computed prefix tree of size at most $(\ell \cdot d + 1)^w$, constructed in time complexity $O((\ell \cdot d)^w)$.* □

4.3 Enumeration of anchor sets

Assume that all patterns have at most width w and depth d. All that remains to turn proposition 8 into a complete solution for pattern matching is to enumerate all possible sets X of at most w vertices in G that are the canonical anchors of some subgraph of G of width w. The bound on the number of such sets (proposition 10) is one of the key stepping stones of this paper that makes ℓ-independent matching possible on port graphs.

Procedure. We introduce AllAnchors, a procedure similar to CanonicalAnchors of listing 1, described in listing 2 in detail. AllAnchors takes as input a port graph G, a root vertex r_G and a width $w \geq 1$, and returns all sets of w vertices that form the canonical anchors of some subgraph of G with CT representation rooted at r_G. The main difference between listings 1 and 2 is that the successive calls to ConsumePath on line 35 of listing 1 are replaced by a series of nested loops (lines 42–48 in listing 2) that exhaustively iterate over the possible outcomes for different subgraphs of G. The results of every possible combination of recursive calls are then collected into a list of anchor sets, which is returned.

Proposition 9 (Correctness of AllAnchors)**.** *Let G be a port graph and $H \subseteq G$ be a connected convex subgraph of G of width w. Let r be a vertex of H. We have* CanonicalAnchors$(H, r) \in$ AllAnchors(G, r, w).

The proof is by induction over the width w of the subgraph H and given in appendix A.4. The idea is to map every recursive call to ConsumePath in listing 1 to a call to AllConsumePath in lines 42–48 of listing 2. All recursive results are concatenated on line 47, and thus the value returned by ConsumePath will be one of the anchor sets in the list returned by AllConsumePath.

We will see that the overall runtime complexity of AllAnchors can be easily derived from a bound on the size of the returned list. For this we use the following result:

Proposition 10. *For a port graph G and vertex r_G in G, the length of the list* AllAnchors(G, r_G, w) *is in $O(c^w \cdot w^{-3/2})$, where $c = 6.75$ is a constant.*

Proof. Let C_w be an upper bound for the length of the list returned by a call to AllConsumePath for width w, and thus a bound on the length of the list returned by AllAnchors. For the base case $w = 0$, $C_0 = 1$. The returned `all_anchors` list is obtained by pushing anchor lists one by one on line 48. We can count the number of times this line is executed by multiplying the length of the lists returned by the recursive calls on lines 43–45, giving us the recursion relation

$$C_w \leq \sum_{\substack{0 \leq w_1, w_2, w_3 < w \\ w_1 + w_2 + w_3 = w - 1}} C_{w_1} \cdot C_{w_2} \cdot C_{w_3}. \tag{10}$$

Listing 2: Returns all sets of w vertices that form the canonical anchors of some subgraph of G with CT representation rooted at r_G.. The code structure mirrors listing 1, with AllAnchors and AllConsumePath replacing CanonicalAnchors and ConsumePath respectively. `lp.split_at`, `G.linear_paths` and ++ are defined as in listing 1.

```
function AllAnchors(G: Graph, root: Vertex, w: Integer) -> List[List[Vertex]]:
  # Assumption: root is on a single linear path
  assert len(G.linear_paths(root)) == 1

  # Initialise the variables for AllConsumePath and return the anchor lists
  all_anchors = []
  for (anchors, seen_paths) in AllConsumePath(G, w, [root], ∅):
    all_anchors.push(anchors)
  return all_anchors

function AllConsumePath(
    G: Graph,
    w: Integer,
    path: Queue[Vertex],
    seen_paths: Set[LinearPath],
) → List[(List[Vertex], Set[LinearPath])]:
  # Base case: return one empty anchor list
  if w == 0:
    return [[]]

  new_anchor = null
  unseen = ∅
  # Find the first vertex in the queue on an unseen linear path
  while unseen == ∅:
    if path.is_empty():
      return []
    new_anchor = path.pop()
    unseen = G.linear_paths(new_anchor) \ seen_paths
  # Every vertex is on at most one unseen linear path as either
  #  - the new anchor is the root, in which case it is on at most one linear path
  #  - or it is on up to two linear paths, but one of them has already been seen.
  assert len(unseen) == 1
  new_path = unseen[0]

  # The w anchors are made of the new anchor and w-1 anchors on path1 - path3
  path1 = path
  path2, path3 = new_path.split_at(new_anchor)
  seen0 = seen_paths ∪ {new_path}
  return_list = []
  # Iterate over all ways to split w-1 anchors over the three paths
  # and solve recursively
  for 0 ≤ w1, w2, w3 < w such that w1 + w2 + w3 == w - 1:
    for (anchors1, seen1) in AllConsumePath(G, w1, path1, seen0):
      for (anchors2, seen2) in AllConsumePath(G, w2, path2, seen1):
        for (anchors3, seen3) in AllConsumePath(G, w3, path3, seen2):
          # Concatenate new anchor with anchors from all paths
          anchors = [new_anchor] ++ anchors1 ++ anchors2 ++ anchors3
          return_list.push(anchors, seen3)
  return return_list
```

Since C_w is meant to be an upper bound, we replace $\leq$ with equality in eq. (10) to obtain a recurrence relation for C_w. This recurrence relation is a generalisation of the well-known Catalan numbers [18], equivalent to counting the number of ternary trees with w internal nodes: a ternary tree with $w \geq 1$ internal nodes is made of a root along with three subtrees with w_1, w_2 and w_3 internal nodes respectively, with $w_1 + w_2 + w_3 = w - 1$. A closed form solution to this problem can be found in [1]:

$$C_w = \frac{\binom{3w}{w}}{2w+1} = \Theta\left(\frac{c^w}{w^{3/2}}\right)$$

satisfies eq. (10) with equality, where $c = {}^{27}/_{4} = 6.75$ is a constant obtained from the Stirling approximation:

$$\binom{3w}{w} = \frac{(3w)!}{(2w)!w!} = \Theta\left(\frac{1}{\sqrt{w}}\right)\left(\frac{(3w)^3}{e^3}\right)^w\left(\frac{e^2}{(2w)^2}\right)^w\left(\frac{e}{w}\right)^w = \Theta\left(\frac{({}^{27}/_{4})^w}{w^{1/2}}\right). \qquad \square$$

To obtain a runtime bound for ALLANCHORS, it is useful to identify how much of G needs to be traversed. If we suppose all patterns have at most depth d, then it immediately follows that any vertex in G that is in the image of a pattern embedding must be at most a distance d away from an anchor vertex. For this purpose, we modify the definition of `split_at` in listing 2 to only return the first d vertices of any path returned. We thus obtain the following runtime.

Corollary 11. *For patterns with at most width w and depth d, the total runtime of* ALLANCHORS *is in*

$$O\left(\frac{c^w \cdot d}{w^{1/2}}\right). \tag{11}$$

The proof is in appendix A.5. Finally, we reach our main result.

Theorem 12. *Let $P_1, \ldots, P_\ell$ be patterns with at most width w and depth d. The pre-computation runs in time and space complexity*

$$O\left((d \cdot \ell)^w \cdot \ell + \ell \cdot w^2 \cdot d\right).$$

For any subject port graph G, the pre-computed prefix tree can be used to find all pattern embeddings $P_i \to G$ in time

$$O\left(|G| \cdot \frac{c^w}{w^{1/2}} \cdot d\right) \tag{12}$$

where $c = 6.75$ is a constant.

Proof. The pre-computation consists of running the CANONICALANCHORS procedure on every pattern and then transforming them into string tuples using ASSTRINGS. ASSTRINGS is linear in pattern sizes and CANONICALANCHORS runs in $O(w^2 \cdot d)$ for each pattern (proposition 4). This is followed by the insertion of ℓ tuples of $2w$ strings of length $\Theta(d)$ into a multidimensional prefix tree. This dominates the total runtime, which can be obtained directly from proposition 14.

The complexity of pattern matching itself on the other hand is composed of two parts: the computation of all anchor set candidates, and the execution of the prefix string matcher for each of the trees resulting from these sets of fixed anchors. The complexity of the former is obtained by multiplying the result of proposition 10 with $|G|$, as ALLANCHORS must be run for every choice of root vertex r in G:

$$O(w \cdot d \cdot C_w \cdot |G|), \tag{13}$$

where C_w is the bound for the number of anchor lists returned by ALLANCHORS. For the latter we use proposition 14 and obtain the complexity $O(w \cdot d \cdot C_w)$, which is dominated by eq. (13). $\square$

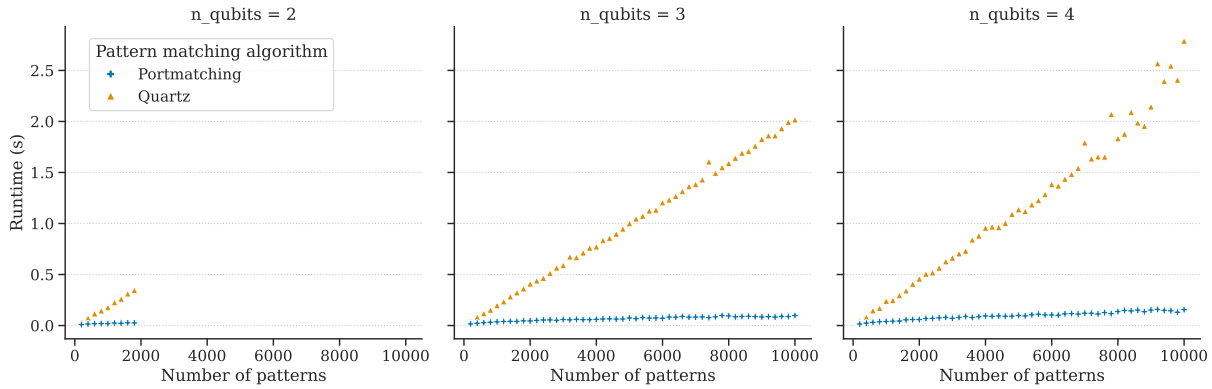


Figure 4: Runtime of pattern matching for $\ell = 0 \ldots 10^4$ patterns on 2, 3 and 4 qubit quantum circuits from the Quartz ECC dataset, for our implementation (Portmatching) and the Quartz project. All $\ell = 1954$ two qubit circuits were used, whereas for 3 and 4 qubit circuits, $\ell = 10^4$ random samples were drawn.

5 Pattern matching in practice

Theorem 12 shows that pattern independent matching can scale to large datasets of patterns but imposes some restrictions on the patterns and embeddings that can be matched. In this section we discuss these limitations and give empirical evidence that the pattern matching approach we have presented can be used on a large scale, outperforming existing solutions.

Pattern limitations. In section 3.2, we imposed conditions on the pattern embeddings in order to obtain a complexity bound for pattern independent matching. We argued how these restrictions are natural for applications in quantum computing and most of the arguments will also hold for a much broader class of computation graphs.

In future work, it would nonetheless be of theoretical interest to explore the importance of these assumptions and their impact on the complexity of the problem. As a first step towards a generalisation, our implementation and all our benchmarks in this section do not make any of these simplifying assumptions. Our results below give empirical evidence that a significant performance advantage can be obtained regardless.

Implementation. We provide an open source implementation in Rust of pattern independent matching using the results of section 4, described in more detail in appendix F. The implementation works for weighted or unweighted port graphs, and makes none of the simplifying assumptions employed in the theoretical analysis.

Benchmarks. To assess practical use, we have benchmarked our implementation against a leading C++ implementation of pattern matching for quantum circuits from the Quartz superoptimiser project [20]. Using a real world dataset of patterns obtained by the Quartz equivalence classes of circuits (ECC) generator, we measured the pattern matching runtime on a random subset of up to 10000 patterns. We considered circuits on the T, H, CX gate set with up to 6 gates and 2, 3 or 4 qubits. Thus for our patterns we have the bound $d \leq 6$ for the maximum depth and width $w = 2, 3, 4$. In all experiments the graph G subject to pattern matching was `barenco_tof_10` input, i.e. a 19 qubit circuit input with 674 gates obtained by decomposing a 10-qubit Toffoli gate using the Barenco decomposition [2]. The results are

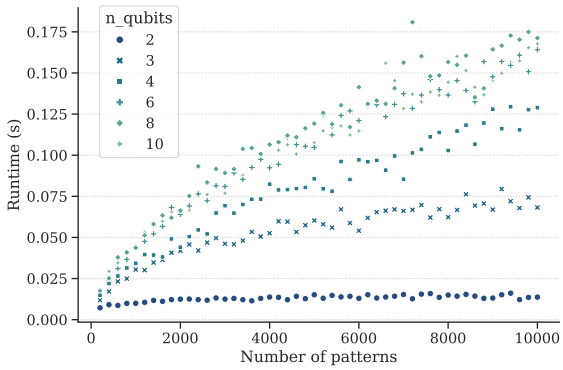


Figure 5: Runtime of our pattern matching for random quantum circuits with up to 10 qubits.

summarised in fig. 4. For $\ell = 200$ patterns, our proposed algorithm is $3\times$ faster than Quartz, scaling up to $20\times$ faster for $\ell = 10^5$.

We also provide a more detailed scaling analysis of our implementation by generating random sets of 10000 quantum circuits with 15 gates for qubit numbers between $w = 2$ and $w = 10$, using the previous gate set; the results are shown in fig. 5. From theorem 12, we expect that the pattern matching runtime is upper bounded by a ℓ-independent constant. Runtime seems indeed to saturate for $w = 2$ and $w = 3$ qubit patterns, with an observable runtime plateau at large ℓ. From the exponential c^w dependency in eq. (12), it is however to be expected that this upper bound increases rapidly for qubit counts $w \geq 4$. A runtime ceiling is not directly observable at this experiment size but the gradual decrease in the slope of the curve is consistent with the existence of the ℓ-independent upper bound predicted in theorem 12.

6 Conclusion

We have demonstrated that pattern matching on port graphs can be done in a runtime asymptotically independent of the number of patterns by pre-computing an automaton-like data structure. As a result, we obtain a provable computational advantage in the regime of numerous low-width patterns. This opens up promising avenues for graph rewriting and particularly for the optimisation of computation graphs and quantum circuits. Benchmarks further show that the approach is fast in practice. At the scale of interest (10000 pattern circuits with 3-4 qubits), the resulting implementation of pattern matching on quantum circuits is 20x times faster than that of Quartz [20], a leading quantum superoptimiser.

References

[1] Jean-Christophe Aval (2008): *Multivariate Fuss–Catalan numbers*. *Discrete Mathematics* 308(20), p. 4660–4669, doi:10.1016/j.disc.2007.08.100.

[2] Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin & Harald Weinfurter (1995): *Elementary gates for quantum computation*. *Phys. Rev. A* 52, pp. 3457–3467, doi:10.1103/PhysRevA.52.3457.

[3] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski & Fabio Zanasi (2020): *String Diagram Rewrite Theory I: Rewriting with Frobenius Structure*. *Journal of the ACM (JACM)* 69, pp. 1 – 58, doi:10.1145/3502719.

[4] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski & Fabio Zanasi (2021): *String diagram rewrite theory II: Rewriting with symmetric monoidal structure*. Mathematical Structures in Computer Science 32, pp. 511 – 541, doi:10.1017/S0960129522000317.

[5] Jin Fang, Yanyan Shen, Yue Wang & Lei Chen (2020): *Optimizing DNN computation graph using graph substitutions*. In: Proceedings of the VLDB Endowment, 13, pp. 2734 – 2746, doi:10.14778/3407790.3407857.

[6] Maribel Fernández, Hélène Kirchner & Bruno Pinaud (2018): *Labelled Port Graph – A Formal Structure for Models and Computations*. Electronic Notes in Theoretical Computer Science 338, pp. 3–21, doi:10.1016/j.entcs.2018.10.002. The 12th Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2017).

[7] Charles L. Forgy (1982): *Rete: A fast algorithm for the many pattern/many object pattern match problem*. Artificial Intelligence 19(1), pp. 17–37, doi:10.1016/0004-3702(82)90020-0.

[8] Zhihao Jia, Oded Padon, James J. Thomas, Todd Warszawski, Matei A. Zaharia & Alexander Aiken (2019): *TASO: optimizing deep learning computation with automatic generation of graph substitutions*. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles, doi:10.1145/3341301.3359630.

[9] Xiaoyi Jiang & Horst Bunke (1996): *Including geometry in graph representations: A quadratic-time graph isomorphism algorithm and its applications*. In Petra Perner, Patrick Wang & Azriel Rosenfeld, editors: Advances in Structural and Syntactical Pattern Recognition, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 110–119, doi:10.1007/3-540-61577-6_12.

[10] Xiaoyi Jiang & Horst Bunke (1998): *Marked subgraph isomorphism of ordered graphs*. In Adnan Amin, Dov Dori, Pavel Pudil & Herbert Freeman, editors: Advances in Pattern Recognition, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 122–131, doi:10.1007/BFb0033230.

[11] Donald Knuth (1999): *The Art of Computer Programming: Sorting and Searching, Volume 3*. Addison-Wesley, Reading MA.

[12] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache & Oleksandr Zinenko (2021): *MLIR: Scaling Compiler Infrastructure for Domain Specific Computation*. In: 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 2–14, doi:10.1109/CGO51591.2021.9370308.

[13] Bruno T. Messmer & Horst Bunke (1999): *A decision tree approach to graph and subgraph isomorphism detection*. Pattern Recognit. 32, pp. 1979–1998, doi:10.1016/S0031-3203(98)90142-X.

[14] Adam Paszke, Sam Gross, Francisco Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai & S. Chintala (2019): *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. In: Neural Information Processing Systems, doi:10.5555/3454287.3455008.

[15] Xuguang Ren & Junhu Wang (2016): *Multi-Query Optimization for Subgraph Isomorphism Search*. Proc. VLDB Endow. 10(3), p. 121–132, doi:10.14778/3021924.3021929.

[16] Timos K. Sellis (1988): *Multiple-Query Optimization*. ACM Trans. Database Syst. 13(1), p. 23–52, doi:10.1145/42201.42203.

[17] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington & Ross Duncan (2020): *tket: a retargetable compiler for NISQ devices*. Quantum Science and Technology 6(1), p. 014003, doi:10.1088/2058-9565/ab8e92.

[18] Richard P. Stanley (2015): *Catalan Numbers*. Cambridge University Press, doi:10.1017/CBO9781139871495.

[19] Amanda Xu, Abtin Molavi, Lauren Pick, Swamit Tannu & Aws Albarghouthi (2023): *Synthesizing Quantum-Circuit Optimizers*. Proc. ACM Program. Lang. 7(PLDI), doi:10.1145/3591254.

[20] Mingkuan Xu, Zikun Li, Oded Padon, S. Lin, J. Pointing, A. Hirth, H. Ma, J. Palsberg, A. Aiken, U.A. Acar & Z. Jia (2022): *Quartz: Superoptimization of Quantum Circuits*. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2022, Association for Computing Machinery, New York, NY, USA, p. 625–640, doi:10.1145/3519939.3523433.

A Proofs

A.1 Proof of proposition 2

Proposition 2. *Let G be a port graph with n_{odd} vertices of odd degree and n_ω open ports. Then the graph width of G is at most $\lfloor (n_{odd} + n_\omega)/2 \rfloor$.*

Proof. For any acyclic linear path $P \subseteq E^*$ in G consider its two endpoints v_1 and v_2, i.e. the two vertices in G that are only incident to one edge in P (linear paths are never empty). Let p_1 and p_2 be the ports where the first and last edges are attached to v_1 and v_2 respectively. Let $p_i' \in ports(v_i)$ such that $p_i' \sim p_i$ for $i = 1,2$. By eq. (3), either $\lambda(v_i, p_i') \in P$ or $\lambda(v_i, p_i') = \omega$. By injectivity of $\lambda(v_i, \cdot)$, the first case implies $p_i = p_i'$ as we would otherwise have two edges $\lambda(v_i, p_i') \neq \lambda(v_i, p_i)$ in P incident to v_i. We thus conclude that either $\lambda(v_i, p_i') = \omega$ or p_i is in a singleton equivalence class of $\sim$ in $ports(v_i)$.

There are n_ω pairs $(v, p) \in V(G) \times \mathscr{P}$ such that $\lambda(v_i, p_i') = \omega$ and n_{odd} pairs $(v, p) \in V(G) \times \mathscr{P}$ such that the equivalence class of p is a singleton set in $ports(v)$. We conclude that there can be at most $n_{\text{odd}} + n_\omega$ end ports of linear paths in G. As every linear path has two end ports and every end port must be distinct, the result follows. □

A.2 Proof of proposition 4

Proposition 4. *For a connected port graph G of width w and depth d and for a vertex r of G, listing 1 returns an anchor set of w vertices; we call this the set of canonical anchors. Its runtime is $O(w^2 \cdot d)$.*

Proof. Termination: we count the number of times CONSUMEPATH is called in one execution of CANONICALANCHORS. The call on line 3 happens exactly once, so we can ignore it. On the other hand, the `path` argument to CONSUMEPATH will always be distinct between any two calls on line 33: it is either a path on a previously unseen linear path, or it is a strict subset of the `path` argument passed to the current call. As there are w linear path with at most d vertices, there is a finite number of calls to CONSUMEPATH. The while loop on lines 14–18 pops an element from the `path` queue at each iteration, so can only be executed a finite number of times. Thus we can conclude that the CONSUMEPATH procedure always terminates.

Correctness: CANONICALANCHORS returns w vertices: in every call to CONSUMEPATH, the only non-recursive insertion to the list of anchors is the initialisation of the `anchors` list on line 31. This insertion happens if and only if `unseen` is non-empty (line 14). Using the assumption that every vertex is on at most 2 linear paths, we can furthermore restrict ourselves to $|\texttt{unseen}| = 1$. Thus the size of `seen_paths` increases by one at every recursion (line 21). The size of `seen_paths` is bounded by w and thus w anchors are added to the `anchors` list over the execution of CANONICALANCHORS.

Let X be the vertices returned by CANONICALANCHORS as a set. It remains to be shown that the X-split graph of G is connected and acyclic. A cycle C in $\text{split}_X(G)$ must have edges on at least 2 distinct linear paths by the second assumption of section 3.2. Say there are $k > 2$ distinct linear paths on the cycle. Every anchor vertex on the cycle can be on either 1 or 2 linear paths (we assumed in section 3.2 that no vertex is on more than two paths). There must be at least k anchor vertices on C whose two adjacent edges that are also in C are on two different linear paths—one for every "switch" of linear path on C. However, by line 14, for every anchor there is at least one unseen linear path, so for k anchors there must be at least $k+1$ linear paths in C, which is impossible.

For connectedness, observe that for every vertex v in the split graph there is a path from the root r to v. In G, such a path is obtained by following the graph traversal implicit in the calls to CONSUMEPATH:

let $\tilde{v}$ be the vertex in G that when split generates v. Every vertex in G appears in the `path` argument to CONSUMEPATH at least once. There is thus an anchor $a \in X$ with a path along a linear path from a to $\tilde{v}$. Applying this argument recursively, there is a sequence of anchors $r = a_1, a_2, \ldots, a_k = \tilde{v}$ corresponding to successive calls to CONSUMEPATH such that for all $1 \leq i < k$ there is a linear path between a_i and a_{i+1}.

We show that the path from r to $\tilde{v}$ through $a_1, \ldots, a_k$ is mapped onto a path in the split graph. In other words, we need to show that the edges along the path are rewired in such a way that adjacent edges along the path are mapped to edges adjacent to the same split vertex. We partition the path into sections from a_i to a_{i+1} for $i = 1, \ldots, k-1$ and consider each subpath separately. Let $e_1, \ldots, e_m$ be the edges of the subpath from a_i to a_{i+1}. The first edge e_1 on this subpath is always in the split graph as $a_i \in X$ and thus is not split. Every other edge, on the other hand, is on the same linear path as e_1. Thus for $1 \leq j < m$, if e_j ends in port p and the next edge e_{j+1} starts in port p', then $p \sim p'$. Thus both edges are mapped to the same split vertex, concluding that the path from r to v is also a path in the split graph.

Complexity: Note that a recursive call to CONSUMEPATH (line 33) occurs if and only if the current call is adding a new element to the list of anchor vertices (line 31). Since we have previously established that the number of anchor vertices returned by CANONICALANCHORS is w, it follows that there are at most w recursive calls to CONSUMEPATH. Therefore, to prove the runtime complexity $O(w^2 \cdot d)$ of CANONICALANCHORS, it remains to show that the execution of the body of CONSUMEPATH—excluding line 33—runs in $O(w \cdot d)$. This is straightforward to check for all lines but 18 and 26. Line 26 is executed at most w times on a single call to CONSUMEPATH, and since `lp.split_at(v)` simply needs to traverse a linear path of at most length d, the required runtime complexity holds. On the other hand, line 18 is executed at most d times on a single call to CONSUMEPATH—since that is what it takes for line 17 to pop all elements from the path.

Assuming the list of linear paths was computed in advance (in time $O(w \cdot d)$, before the first call to CONSUMEPATH in line 3) and each linear path is given a unique index $0 \ldots w-1$, we can store the `seen_paths` set and the set of linear paths for each vertex as ordered lists of linear path indices. The corresponding set `G.linear_paths(v)` can be stored as an attribute of `v` and be retrieved in constant time (assuming for instance that vertices are indexed from 0 to $|G|-1$). Other than that, line 18 is a set operation that can be realised in a single $O(w)$ pass over the ordered lists `unseen_paths` and `G.linear_paths(v)` since both have at most w elements. □

A.3 Proof of proposition 7

Proposition 7. *Let T_1, T_2 be acyclic connected split graphs of width w and let r_1, r_2 be vertices in T_1 resp. T_2. Let $(s_1, \ldots, s_{2w}) = \text{ASSTRINGS}(T_1, r_1)$ and $(t_1, \ldots, t_{2w}) = \text{ASSTRINGS}(T_2, r_2)$ be the string encodings of their linear paths. Then there is an injective tree embedding $T_1 \to T_2$ that satisfies eq.* (4)*, maps r_1 to r_2 and preserves vertex labels if and only if $s_i \subseteq t_i$ for all $1 \leq i \leq 2w$.*

Proof. The $\Rightarrow$ direction is straightforward. By assumption, the root r_1 in T_1 is mapped to the root r_2 in T_2. Non-root anchors on the other hand are precisely the vertices on more than one path in T_1 and T_2. If $\varphi : T_1 \to T_2$ is an injection of trees of the same width, then all non-root anchors of T_1 must be mapped to the non-root anchors of T_2. Every linear path in T_1 includes at least one anchor vertex. This must be mapped by φ to a path in T_2, through the corresponding anchor vertex in T_2. As φ preserves the port labels (eq. (4)), the image path in T_2 must be a subpath of a linear path of T_2. As the linear paths are split and ordered starting from anchor vertices, the string encoding of every split linear path in T_1 will be a prefix of the string encodings of split linear paths in T_2.

$\Leftarrow$: it suffices to show that every path from root to a vertex in T_1 is also a path from root to a vertex in T_2 and that the vertex labels coincides. A path P from root r_1 in T_1 can be partitioned into a sequence of paths $P = P_1 \cdots P_k$, which all start at anchors and are subpaths of linear paths of T_1. These subpaths corresponds to a sequence of prefixes of $s_{\alpha_1}, s_{\alpha_k}$ in the string encoding, which are also prefixes of $t_{\alpha_1}, \ldots, t_{\alpha_k}$. Since the vertex labels are stored in the tuple string encoding, we know that the end vertex of the path $P_1 \cdots P_i$ coincides with the anchor of $t_{\alpha_{i+1}}$ in T_2. Applying this argument recursively on the chain of linear subpaths, we conclude that $P = P_1 \cdots P_k$ is also a path in T_2. Finally, the vertex labels must coincide on the shared domain of definition, as the string encoding coincide. Equation (4) can be shown to be satisfied using a similar argument to the one presented in the proof of proposition 6. □

A.4 Proof of proposition 9

Proposition 9 (Correctness of ALLANCHORS)**.** *Let G be a port graph and $H \subseteq G$ be a connected convex subgraph of G of width w. Let r be a vertex of H. We have* CANONICALANCHORS$(H, r) \in$ ALLANCHORS(G, r, w).

Proof. Let $H \subseteq G$ be a connected subgraph of G of width w. We prove inductively over w that

$$\text{CONSUMEPATH}(H, \texttt{path}, \texttt{seen\_paths}) \in \text{ALLCONSUMEPATH}(G, w, \texttt{path}, \texttt{seen\_paths}) \tag{14}$$

for all arguments `path` and `seen_paths`. The statement in the proposition follows from this claim directly.

For the base case $w = 1$, CONSUMEPATH will return `[new_anchor]`, where `new_anchor` is obtained from lines 16–20 of listing 1: there is only one linear path and thus for every recursive call to CONSUMEPATH, `unseen` will be empty, until `path` has been exhausted and the empty list is returned. The definition of `new_anchor` coincides with the one obtained from lines 20–28 of listing 2. The only values of `w1`, `w2` and `w3` that satisfy the loop condition on line 42 of listing 2 for $w = 1$ are $\texttt{w1} = \texttt{w2} = \texttt{w3} = 0$. Using the base condition on lines 18–20 of listing 2, we conclude that ALLCONSUMEPATH$(G, 1, \texttt{path}, \texttt{seen_paths})$ returns `[[new_anchor]]`, satisfying eq. (14).

We now prove the claim for $w > 1$ by induction. Using our simplifying assumptions, we obtain the assertion on line 32 of listing 2, as documented. For listing 1, this assumption simplifies the loop on lines 34–37 to at most three calls to CONSUMEPATH with arguments (H, P_{curr}, S_{curr}), (H, P_ℓ, S_ℓ) and (H, P_r, S_r) respectively, where

- P_{curr} is the value of the `path` variable after line 20,
- P_ℓ and P_r refer to the two halves of the new linear path, as computed and stored in the variables `left_path` and `right_path` on line 28, and
- S_{curr}, S_ℓ and S_r are the values of the `seen_paths` variable after the successive updates on line 23 and two iterations of line 37.

Consider a call to CONSUMEPATH (listing 1) with arguments $\texttt{G} = H$ and some variables `path` and `seen_paths`. Let w_{curr}, w_ℓ and w_r be the length of the values returned by the three recursive calls to CONSUMEPATH of line 35. As every anchor vertex reduces the number of unseen linear paths by exactly one (using the simplifying assumptions), it must hold that $w_{curr} + w_\ell + w_r + 1 = w$. Thus for a call to ALLCONSUMEPATH (listing 2) with arguments $\texttt{G} = G$, $\texttt{w} = w$ and the same values for `path` and `seen_paths`, there is an iteration of the `for` loop on line 42 of listing 2 such that $\texttt{w1} = w_{curr}, \texttt{w2} = w_\ell$ and $\texttt{w3} = w_r$. The definition of `seen0` on line 38 of listing 2 coincides with the update to `seen_paths` on line 23 of listing 1; it follows that on line 43 of listing 2 the recursive call ALLCONSUMEPATH$(G, w_{curr}, P_{curr}, S_{curr})$ is

executed. From the induction hypothesis we obtain that there is an iteration of the `for` loop on line 43 of listing 2 in which `anchors1` and `seen1` coincide with the `new_anchors` and `new_seen_paths` variables of the first iteration of the `for` loop on line 34 of listing 1. In particular the value of `seen1` is equal S_ℓ.

Repeating the argument, we obtain that there are iterations of the `for` loops on lines 44 and 45 of listing 2 that correspond to the second and third calls to CONSUMEPATH on line 35 of listing 1. Finally, the concatenation of anchor lists on line 47 of listing 2 is equivalent to the repeated concatenations on line 36 of listing 1 and so we conclude that eq. (14) holds for w. □

A.5 Proof of corollary 11

Corollary 11. *For patterns with at most width w and depth d, the total runtime of* ALLANCHORS *is in*

$$O\left(\frac{c^w \cdot d}{w^{1/2}}\right). \tag{11}$$

Proof. We restrict `split_at` on line 37 to only return the first d vertices on the linear path in each direction: vertices more than distance d away from the anchor cannot be part of a pattern of depth d.

We use the bound on the length of the list returned by calls to ALLCONSUMEPATH of proposition 10 to bound the runtime. We can ignore the non-constant runtime of the concatenation of the outputs of recursive calls on line 47, as the total size of the outputs is asymptotically at worst of the same complexity as the runtime of the recursive calls themselves. Excluding the recursive calls, the only remaining lines of ALLCONSUMEPATH that are not executed in constant time are the `while` loop on lines 24–28 and the `split_at` call on line 37.

Consider the recursion tree of ALLCONSUMEPATH, i.e. the tree in which the nodes are the recursive calls to ALLCONSUMEPATH and the children are the executions spawned by the nested `for` loops on line 42–48. This tree has at most

$$C_w = \Theta\left(\frac{c^w}{w^{3/2}}\right)$$

leaves. A path from the root to a leaf corresponds to a stack of recursive calls to ALLCONSUMEPATH. Along this recursion path, the `seen_paths` set is always strictly growing (line 38) and the vertices popped from the `path` queue on line 27 are all distinct. `split_at` is called once for each of the w linear path that are added to `seen_paths`. For each linear path two paths of length at most d are traversed and returned. Thus the total runtime of `split_at` along a path from root to leaf in the recursion tree is in $O(w \cdot d)$. Similarly, the number of executions of the lines 25–28 is bound by the number of elements that were added to a `path` queue, as for every iteration an element is popped off the queue on line 27. This is equal to the number of elements returned by `split_at`, resulting in the same complexity. We can thus bound the overall complexity of executing the entire recursion tree by $O(C_w \cdot w \cdot d) = O(\frac{c^w \cdot d}{w^{1/2}})$. □

B Lower bound on the number of patterns

Proposition 13. *Let $N_{w,d}$ be the number of port graphs of width w, depth d and maximum degree $\Delta \geq 4$. We can lower bound*

$$N_{w,d} > \left(\frac{w}{2e}\right)^{\Theta(wd)},$$

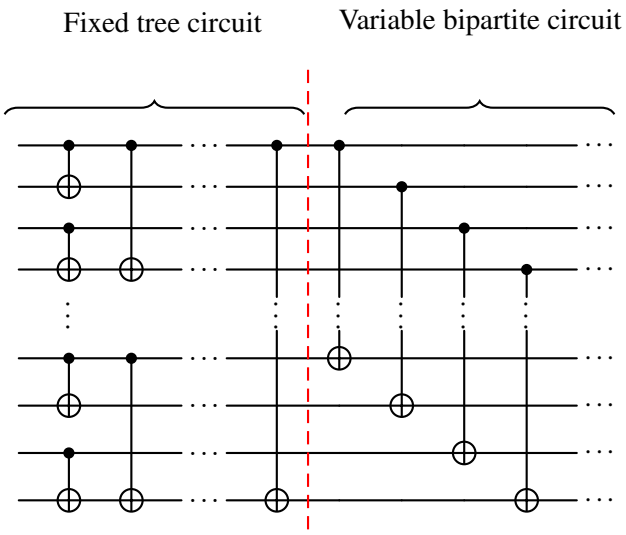
assuming $w \leq o(2^d)$.

In the regime of interest, w is small, so the assumption $w \leq o(2^d)$ is not a restriction. In the main text we use the bound $|P| \leq w \cdot d$ to avoid introducing the circuit depth. The bound stated in eq. (2) is thus slightly looser.

Proof. Let $w, d > 0$ and $\Delta \geq 4$ be integers. We wish to lower bound the number of port graphs of depth d, width w and maximum degree Δ. It is sufficient to consider a restricted subset of such port graphs, whose size can be easily lower bounded. We will count a subset of CX quantum circuits, i.e. circuits with only *CX* gates, a two-qubit non-symmetric gate. Because we are using a single gate type, this is equivalent to counting a subset of port graphs with vertices of degree 4. Assume w.l.o.g that w is a power of two. We consider CX circuits constructed from two circuits with w qubits composed in sequence:

- **Fixed tree circuit**: A $\log_2(w)$-depth circuit that connects qubits pairwise in such a way that the resulting port graph is connected. We fix such a tree-like circuit and use the same circuit for all CX circuits. We can use this common structure to fix an ordering of the w qubits, that refer to as qubits $1, \ldots, w$.
- **Bipartite circuit**: A CX circuit of depth $D = d - \log_2(w)$ with exactly $w/2 \cdot (d - \log_2(w))$ CX gates, each gate acting on a qubit $1 \leq q_1 \leq w/2$ and a qubit $w/2 < q_2 \leq w$.

The following circuit illustrates the construction:



All that remains is to count the number of such bipartite circuits. Every slice of depth 1 must have $w/2$ CX gates acting on distinct qubits. Every qubit 1 to $w/2$ must interact with one of the qubits $w/2+1$ to w, so there are $(w/2)!$ such depth 1 slices. Repeating this depth 1 construction D times and using Sterling's approximation, we obtain a lower bound for the number of port graphs of depth d, width w and maximum degree at least 4:

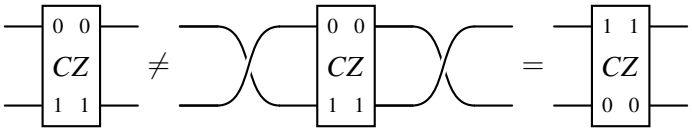
$$\left(\left(\frac{w}{2}\right)!\right)^D > \sqrt{w\pi}\left(\frac{w}{2e}\right)^{wD/2} = \left(\frac{w}{2e}\right)^{\Theta(w\cdot d)}$$

where we used $w = o(2^d)$ to obtain $\Theta(D) = \Theta(d)$ in the last step. □

C Quantum circuits as port graphs

A relevant consideration when viewing quantum circuits as port graphs is the question of equality on circuits. We consider two circuits to be equal if they are equal as port graphs. This sense of equality is more general than equality of ordered lists of gates, another common internal representation of quantum computations, but does not account for commuting gates or gate *symmetries*. An example of a symmetric gate type is the CZ gate, a gate type of arity $n = 2$ that is symmetric in its arguments

that is to say, exchanging the order of the inputs and outputs does not change the computation. Viewed as port graphs, however, the left and right hand side are distinct circuits



In the case that such symmetries need to be taken into account for pattern matching, there are two simple solutions. For rewriting purposes, one may choose to add a single rewrite rule to express the symmetry explicitly, stating that the symmetric gate can be rewritten to itself with the edge order reversed. This will recover the full expressivity of the rewrite rule set, at the expense of additional rewrite rule applications.

Alternatively, all instances of a pattern that are equivalent up to gate symmetries can be enumerated and added as separate patterns to the matcher. This approach is particularly appealing as the runtime of the pattern matcher will remain unchanged, despite the increase in the number of patterns (exponential in the number of symmetric gates). The trade-off is increased pre-compilation time and pattern matcher size.

D Properties of the Canonical Tree representation

We provide here the exact derivations of the properties of the CT representation that we rely on, namely an injective map from the port graph representation to CTs, invariance of the CT representation under pattern embeddings and the string encoding of CT trees.

Equivalence of the CT representation. A connected port graph G is fully defined by the set of edges, given as a set of pairs in $V \times \mathscr{P}$. Given the CT representation of G with vertices $\tilde{V}$, alongside a map $merge : \tilde{V} \to V$ that maps the vertices of the CT representation to the vertices of G, it is immediate that G can be recovered by mapping every $(v, p) \in \tilde{V} \times \mathscr{P}$ to $(merge(v), p) \in V \times \mathscr{P}$.

Up to isomorphism in the co-domain V, we can store *merge* by storing the partition of $\tilde{V}$ into sets with the same image. We introduce for this a map $\tilde{V} \to \tilde{V}$ that maps every vertex to a canonical representative of the partition—for instance the vertex closest to the root in CT. This map can be stored as vertex labels of CT, which we can refer to as the labelled CT representation for distinction. However in the main text, it is always the labelled representation that is meant when CT representations are discussed.

We thus have a bijective map between the labelled CT representation of G and the port graph G. Furthermore, this map preserves the linear paths, i.e. it maps one to one the linear paths of the labelled CT representation to the linear paths of G. For all purposes, we can thus treat the labelled CT representation as an equivalent representation of G.

Invariance under pattern embedding. Unlike graphs, rooted trees can be defined in a way that is invariant under bijective relabelling of the vertices by using the invariant port labels. Every tree vertex is either the root vertex or it is uniquely identified by the path to it from the root. Since paths can be defined in terms of port labels, paths are invariant under pattern embeddings of the underlying graph.

For trees T and T', let S and S' be the sets of their respective vertices expressed as sequences of port labels. We thus define tree inclusion and equality only up to vertex relabelling: $T \subseteq T'$ if and only if $S \subseteq S'$, and $T = T'$ if and only if $S = S'$. On labelled trees, we also require inclusion (resp. equality) of the vertex label maps, including in particular the *merge* map of labelled CT representations. As a result, subtree relations in labelled CT representations correspond to subgraphs of the original graph, in effect reducing the pattern matching problem on port graphs to a problem of tree inclusion on CT representations. This statement is formalised in proposition 6.

String encoding of CT representations. In order for our string encoding of CT representations to map tree inclusion to string prefixes, we recall that the anchor set in eq. (8) is fixed: a subtree of T with the same anchor set can only be obtained by shortening the linear paths at their ends— the resulting subpath will always contain the anchor vertex. Given a linear path L of T, we thus split L at the anchor on L and obtain two paths L_1, L_2 starting from the anchor to the ends of L. For any subtree $T' \subseteq T$, the linear path L' that is a subpath of L will split into L'_1, L'_2, prefixes of L_1 and L_2 respectively.

With an appropriate string representation of CT vertices and their labels, this will encode all linear paths. In the same way that the *merge* map of the labelled CT representation is used to restore the original graph from the split CT vertices, we use it to recover the anchor vertices from the split linear paths. Finally, to order the linear paths in the string tuple, we use for instance the order of their anchors induced by port ordering.

E Prefix Trees

Our main result is achieved by reducing a tree inclusion problem to the following problem.

String prefix matching. Consider the following computational problem over strings. Let Σ be a finite alphabet and consider $\mathscr{W} = (\Sigma^*)^w$ the set of w-tuples of strings over Σ. For a string tuple $(s_1, \ldots, s_w) \in \mathscr{W}$ and a set of string tuples $\mathscr{D} \subseteq \mathscr{W}$, the w-dimensional string prefix matching consists in finding the set

$$\{(p_1, \ldots, p_w) \in \mathscr{D} \mid \text{for all } 1 \leq i \leq w : p_i \text{ is a prefix of } s_i\}.$$

This string problem can be solved using a w-dimensional prefix tree. We give a short introduction to prefix trees for the string case but refer to standard literature for more details [11].

One-dimensional prefix tree. Let $P_1, \ldots, P_\ell \in \mathscr{A}^*$ be strings on some alphabet $\mathscr{A}$. Given an input string $s \in \mathscr{A}^*$, we wish to find the set of patterns $\{P_{1 \leq i \leq \ell} | P_i \subseteq s\}$, i.e. P_i is a prefix of s.

The prefix tree of $P_1, \ldots, P_\ell$ is a tree with a tree node for each prefix of a pattern. The children of an internal node are the strings that extend the prefix by one character. The root of the tree is the empty string. Each tree node also stores a list of matching patterns, with each pattern stored in the unique corresponding node. Every prefix tree has an empty string node, which is the root of the tree. For every inserted pattern of length at most L nodes are inserted, one for every non-empty prefix of the pattern. Thus a one-dimensional prefix tree has at most $\ell \cdot L + 1$ nodes and can be constructed in time $O(\ell \cdot L)$.

Given an input $s \in \mathscr{A}^*$, we can find the set of matching patterns by traversing the prefix tree of $P_1, \ldots, P_\ell$ starting from the root. We report the list of matching patterns at the current node and move to

the child node that is still a prefix of s, if it exists. This procedure continues until no more such child exists. In total the traversal takes time $O(|s|)$, as every character of s is visited at most once.

Note that in theory the number of reported pattern matches can dominate the runtime of the algorithm. We can avoid this by returning the list of matches as an iterator, stored as a list of pointers to the tree nodes matching lists.

Multi-dimensional prefix tree. A w-dimensional prefix tree for $w > 1$ is defined recursively as a one-dimensional prefix tree that at each node stores a $w-1$-dimensional prefix tree. Given an input w-tuple $(s_1, \ldots, s_w) \in (\mathscr{A}^*)^w$, the traversal of the w-dimensional prefix tree is done by traversing the one-dimensional prefix tree on the input s_1 until no child is a prefix of the input, and then recursively traversing the $w-1$-dimensional prefix tree on $(s_2, \ldots, s_w)$. Similarly to the one-dimensional case, the list of matching patterns is stored at prefix tree nodes and reported during traversal. The traversal thus takes time $O(|s_1| + \cdots + |s_w|)$, as every character of s is visited at most once.

For ℓ tuples of size w of words of maximum length L, we can bound the number of nodes of the w-dimensional prefix tree by $1 + (\ell \cdot L)^w$. The runtime and space complexity of the construction of the w-dimensional prefix tree is thus in $O((\ell \cdot L)^w)$, summarised in the result:

Proposition 14. *Let $\mathscr{D} \subseteq \mathscr{W}$ be a set of string tuples and L the maximum length of a string in a tuple of $\mathscr{D}$. There is a prefix tree with at most $(\ell \cdot L)^w + 1$ nodes that encodes $\mathscr{D}$ that can be used to solve the w-dimensional string prefix matching problem in time $O(|s_1| + \cdots + |s_w|)$.*

F Open source implementation

The code is available at `https://github.com/lmondada/portmatching/`. All benchmarking can be reproduced using the tooling and instructions at `https://github.com/lmondada/portmatching-benchmarking`.

We represent all the pattern matching logic within a generalised finite state automaton, composed of states and transitions. This formalism is used to traverse the graph input and express both the prefix tree of the string prefix matching problem and the (implicit) recursion tree of listing 2 in section 4.3. We sketch here the automaton definition. Further implementation details can be obtained from the `portmatching` project directly.

In the pre-computation step, the automaton is constructed based on the set of patterns to be matched. It is then saved to the disk; a run of the automaton on an input graph G is the solution the pattern independent matching problem for the input G. To run the automaton, we keep track of the set of current states, initialised to a singleton root state and updated following allowed transitions from one of the current states. Which transitions are allowed is computed using predicates on the input graph stored at the transitions. This is repeated until no further allowed transitions exist from a current state.

At any one state of the automaton, zero, one or several transitions may be allowed depending on the input graph. As the automaton is run for a given input graph G, we keep track of the vertices that have been matched by the automaton so far with an injective map between a set of unique symbols and the vertices of G. Vertices in this map are the known vertices of G. There are three main types of transitions:

- A **constraint** transition asserts that a property of the known vertices holds. This can be checking for a vertex or edge label, or checking that an edge between two known vertices and ports exists.
- A **new vertex** transition asserts that there is an edge between a known vertex v at a port p and a new vertex at a port p'. The new vertex must not be any of the known vertices. When the transition is followed, a new symbol is introduced and the vertex is added to the symbol vertex map.

- A **set anchor** transition is an ε-transition, i.e. a non-deterministic transition that is always allowed. Semantically, it designates a known vertex as an anchor.

By requiring that all constraint transitions from a given state assert mutually exclusive predicates (such as edges starting from a given vertex and port, or the vertex label of a given vertex), we can ensure that constraint transitions are always deterministic. New vertex transitions are also deterministic in finite depth patterns [5], so that in the regime explored in this paper, the only source of non-determinism is the choice of anchors. Intuitively, this corresponds to the facts that the prefix tree traversal of section 4 is deterministic while the anchors enumeration of listing 2 returns a multitude of options to be explored exhaustively.

To obtain a set of matching patterns from a run of the automaton, we store pattern matches as lists at the automaton states. When a state is added to the set of current states, its list of matches are added to the output. To build the automaton, we consider one pattern at a time, convert it into a chain of transitions of the above types that is then added to the state transition graph. At the target state of the last transition, we then add the pattern ID to the list of matched patterns.

[5] In cyclic and non-convex cases, it can happen that a vertex is both a known vertex of a large pattern and a new vertex within a smaller subpattern.

Modelling Real-time Systems with Bigraphs

Maram Albalwe
University of Glasgow
Glasgow, UK
University of Tabuk
Tabuk, Saudi Arabia
m.albalwe.1@research.gla.ac.uk

Blair Archibald Michele Sevegnani
University of Glasgow
Glasgow, UK
{blair.archibald, michele.sevegnani}@glasgow.ac.uk

Bigraphical Reactive Systems (BRSs) are a graph-rewriting formalism describing systems evolving in two dimensions: spatially, *e.g.* a person in a room, and non-spatially, *e.g.* mobile phones communicating regardless of location. Despite use in domains including communication protocols, agent programming, biology, and security, there is no support for real-time systems. We extend BRSs to support real-time systems with a modelling approach that uses multiple perspectives to represent digital clocks. We use Action BRSs, a recent extension of BRSs, where the resulting transition system is a Markov Decision Process (MDP). This allows a natural representation of the choices in each system state: to either allow time to pass or perform a specific action. We implement our proposed approach using the BigraphER toolkit, and demonstrate the effectiveness through multiple examples including modelling cloud system requests.

1 Introduction

Bigraphs are a computational model where systems are described based on two types of relationships: spatially using *nesting* (and parallel adjacency) and non-local linking through hyperlinks regardless of locations. Like standard graphs, bigraphs have an equivalent diagrammatic and algebraic representation. Bigraphical Reactive Systems (BRSs) equip bigraphs with a set of reaction rules that specify how a system evolves over time, *i.e.* reaction rules substitute sub-bigraphs with other bigraphs.

The standard theory of bigraphs has been extended to model a wider range of systems *e.g.* stochastic bigraphs [11] assign rates to reaction rules, bigraphs with sharing [21] allow intersecting locations, directed bigraphs [8] associate directions to links, conditional bigraphs [2] add conditions to rules, and probabilistic and action bigraphs [3] support probabilistic and non-deterministic behaviour. Using these extensions, bigraphs have been successfully used to model a variety systems including mixed-reality games [6], cloud systems [19], self-adaptive fog systems [20], sensor network infrastructure [23], and rational agents [4].

An extension that has not been explored is real-time bigraphs that can model systems exhibiting non-deterministic behaviour that is controlled by time constraints. While the non-deterministic aspects of real-time systems could be modelled by action bigraphical reactive systems (ABRSs) in which the underlying Markov Decision Process (MDP) semantics allows for a transition choice at each state, there is currently no notion of *clock constraints* in the theory, *e.g.* specifying that after 3 time units have passed, a given action *must* occur.

We propose a modelling strategy to encode timed systems within ABRSs through the well-known MDP-based digital clock approximation. We introduce a clock, as a new bigraph entity, for each timed entity in the system and we place these in a separate region so all the clocks can be manipulated without polluting the actual system model. This allows one reaction rule to advance all clocks at once, and this reduces the state space overhead of the approach. We mirror the real-time passage by introducing a

J. Endrullis, D. Grzelak, T. Heindel and J. Kosiol (Eds):
GCM 2023 and GCM 2024
EPTCS 417, 2025, pp. 96–116, doi:10.4204/EPTCS.417.6

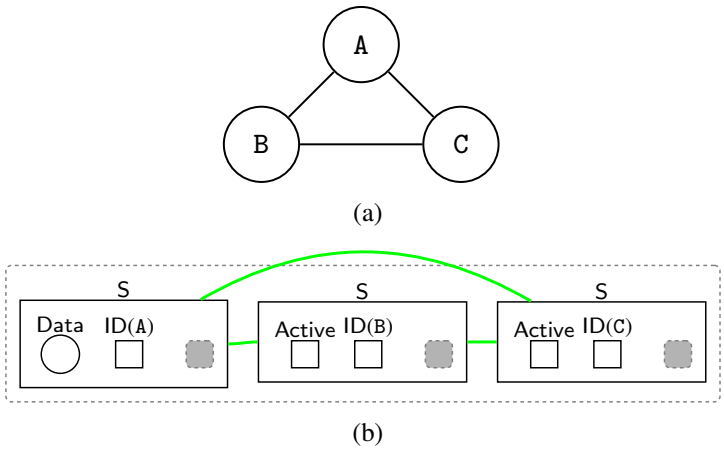


Figure 1: (a) Example network topology with three sensors; (b) corresponding bigraph with data to be transmitted by sensor A and the status of each sensor (*e.g.* Active).

global clock that controls the system execution time, *i.e.* system terminates when we reach the time limit of this clock, and we explicitly model the non-deterministic behaviour, *i.e.* at each state there is a choice between taking an action or allowing time to pass when the constraints are not yet satisfied.

We make the following research contributions:

- We propose a modelling strategy to express clocks within action bigraphs.
- We define a set of reaction rules to model clock constraints in real-time systems through a digital clocks approximation.
- We illustrate how to use our approach in practice by partially modelling the timed aspects of a cloud computing scenario.
- We implement this approach in BigraphER [22] to show this is not just a theoretical contribution, but a practical one.

Outline. We give an informal description of bigraphs and BRSs in Section 2. Section 3 provides a description of non-deterministic models and shows how to formalise digital clocks within ABRSs. We illustrate our approach by providing a BRS implementation for a simple scenario modelled as a (probabilistic) timed automaton and a model of cloud system requests in Section 4. Section 5 gives a brief literature review, and we conclude in Section 6.

2 Background

2.1 Bigraphs

Introduced by Milner [16], Bigraphs provide a powerful diagrammatic representation for modelling systems that evolve in both spatial and non-spatial dimensions. Bigraphs represent the structure of a system through two relations over its entities: a *place graph* that encodes their spatial arrangement, *e.g.* a person in a room, and a *link graph* that specifies, through hyper-edges, non-spatial relations, *e.g.* communication capabilities between network devices. We give an informal description of bigraphs using the example in

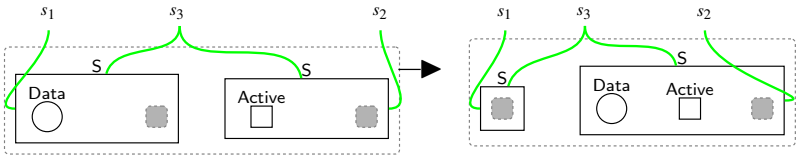


Figure 2: Reaction rule `send_Data` applies when the receiver is active.

Figure 1. A comprehensive formal description can be found elsewhere [16]. Bigraphs have both a diagrammatic representation and *equivalent* algebraic notation. We use the diagrammatic notion throughout this paper.

Figure 1a shows a simple network topology consisting of three sensors connected to each other by communication links. Data may be exchanged between two sensors when the receiver is active. A corresponding bigraph representation is in Figure 1b. Sensors are represented as *entities* of type S that have *nested* identifiers (also just a different type of entity) ID(A), ID(B), and ID(C). The sender contains an entity Data that can be sent to other sensors when they are in the Active mode. We sometimes draw different entity types by using different shapes or colours *e.g.* we have used squares for identifiers and a circles for Data. Entities can be *atomic*, *e.g.* Data, meaning they have no children, or contain any number of other entities. We allow entities to be parameterised to represent families of entities, *i.e.* ID(x) specifies a new entity for every possible value of x (which may be string, integer, or float typed). Sites—filled dashed rectangles—represent unspecified bigraphs: an arbitrary bigraph, including the empty bigraph, might exist there. Bigraphs may consist of more than one region—clear dashed rectangles—which represent adjacent parts of the system. For the link graph, entities have a fixed *arity* that represents the number of *links* they must have (the green edges). Links are hyperlinks that may connect 1-to-n. Open links—a link that has a name— indicates a link that may connect elsewhere, *i.e.* to currently unspecified entities. Links may also be closed, a 1-to-0 hyperedge, which is shown by an orthogonal line at the end of the link. Open names, sites, and regions allow model composition: regions of one bigraph can be placed in the sites of another and like-names joined. This forms the basis of the rewriting theory.

2.2 Bigraphical Reactive Systems (BRSs)

A bigraph represents a system at a single point in time. To model dynamic behaviour, Bigraphical Reactive Systems (BRSs) equip bigraphs with a set of *reaction rules* specifying how the system may evolve. Reaction rules take the form $L \longrightarrow\!\!\!\!\blacktriangleright R$ meaning that when the rule is applicable to a state S (state S *matches* bigraph L) the system evolves by replacing an occurrence of bigraph L in S with bigraph R. BRSs form a transition system that has an initial state (*bigraph*), and the transition from one state (*bigraph*) to another is defined by generating all possible rewrites (*reactions*). For example, the rule in Figure 2 is applicable where there is a sensor that has Data to send to another Active sensor in its range.

We control the execution of a set of reaction rules via *priority classes*. That is $\{\mathtt{r}_1, \mathtt{r}_2\} < \{\mathtt{r}_3\}$ means rules $\mathtt{r}_1$ and $\mathtt{r}_2$ can be applied only if it is not possible to apply $\mathtt{r}_3$. Similarly to entities, rules can be parameterised to allow multiple rule applications over a predefined set of values.

3 Modelling Time with Action Bigraphs

The big idea is that by utilising Action Bigraphs [3] we can encode timed systems. This is based on a well-known approximation of (probabilistic) timed automata to Markov Decision Processes [10]. Intuitively, we extend the usual action semantics to allow two forms of action: *discrete actions* that encode system events (which may only be enabled at some time), and *time* actions that progress time.

Action Bigraphical Reactive Systems (ABRSs) allow a choice of probability distributions at each rewriting step by assigning *weights* to the reaction rules, and by giving action labels to specific sets of reaction rules. An action is enabled/possible whenever there is a reaction rule from the set which is enabled. The resulting transition system is a Markov Decision Processes (MDP) [17] which can be verified using off-the-shelf model checkers such as PRISM [12]. An MDP is a tuple $(S, s_0, A, Step)$ where S is a set of states with a predefined initial state $s_0 \in S$. For each state, we can choose an (enabled) action $a \in A$ which gives an associated probability distribution over future states as specified by $Step$.

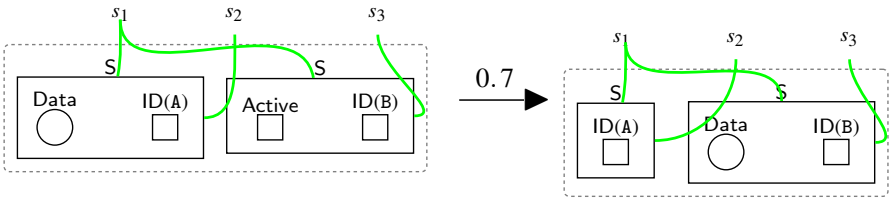
To show how ABRSs model non-deterministic behaviour of real-time systems where underlying transition system is an MDP, we use the bigraph example shown in Figure 1b. That is for sensor `A`, data can either be sent successfully to other sensors or fail to send. We also can permit sensor `A` to send `Data` with a bias by replacing the `send_Data` rule (Figure 2) with the two weighted (as shown in Figure 3). For example, given $weight = 0.7$ to `B` and $weight = 0.3$ to `C`, then sending to `B` is more than twice as likely. We use *bigraphs* throughout this paper to mean *Action Bigraphical Reactive Systems (ABRSs)*. Using BigraphER [22], an open-source toolkit for working with bigraphs (and the supporting ABRSs), we can export the corresponding MDPs transition system to be formally checked. Importantly, action bigraphs still respect any rule priorities. This means an action will fire with any high priority rule before firing with those of lower priorities. We use this feature in our encoding of clock invariants.

While MDPs support uncertainty modelling for dynamic environments, Probabilistic Timed Automata (PTAs) [1] extend this by incorporating timing constraints which offer a more comprehensive framework for modelling and reasoning about real-time systems. A PTA is defined as a tuple in the form (L, l_0, E, A, C, I) where L is a set of locations (*i.e.* states) with $l_0 \in L$ a start location. E is a set of edges that represents the possible transitions between locations while A is a set of (discrete) actions that may occur in the system. C is a finite set of clocks and I is a set of clock invariants associated to each location. While these invariants could be flexible, practically they are typically used to force a transition from a location, *e.g.* an invariant $x \leq 2$ forces a location move at time 2 if we have not already left the location (alongside transition conditions these usually encode a "move within a maximum of 2 time units" semantics). Transitions are associated to actions and probability distributions, with the addition of *clock constraints* (*e.g.* $x > 3$) and *clock resets* (*e.g.* $x := 0$).

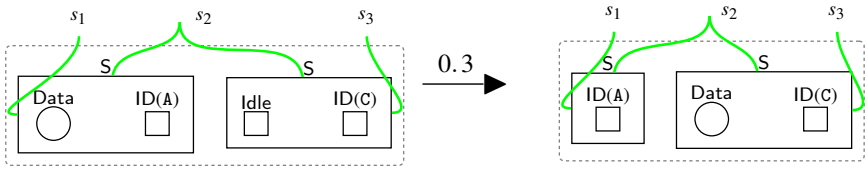
In the *digital clocks approximation* we give the semantics of PTAs as a Markov Decision Process where states (including initial) are all the locations where clock invariants are met, and we form a single action set consisting of both actions manipulating time and user-specified discrete actions as follows:

- a *discrete* action is enabled if all clock constraints (given in E) associated with the transition are satisfied;
- a *time* action is available while invariants associated to $l \in L$ are satisfied as time elapses. That is, we cannot progress time if something must happen beforehand.

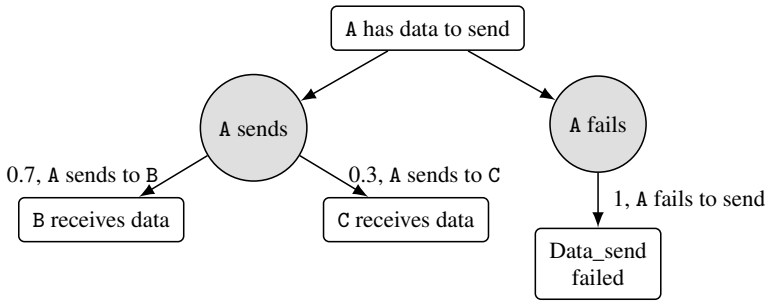
Since we use action bigraphs, we can associate discrete transitions with a probability distribution. If required we can, for example, draw with uniform probability to recover semantics for a non-probabilistic timed automata.



(a) Sensor A sends to sensor B with weight 0.7



(b) Sensor A sends to sensor C with weight 0.3



(c) MDP generated by the ABRS.

Figure 3: Example Action Bigraphical Reactive System: probabilistic reaction rules using weights.

3.1 Digital Clocks in Bigraphs

Standard BRSs evolve to a new state whenever there is a match of a reaction rule[1]. In this sense they are non-deterministic but not explicitly so, *i.e.* we cannot label particular actions without ABRSs. Models of real-time systems need to both find enabled matches, and meet any requirements introduced by clock constraints. We propose a modelling technique that encodes digital clocks as *entities* to model real-time systems. Although PTAs can work with real-valued clocks, for the digital clocks approximation we fix clock values to non-negative integers and bound the total runtime of the system (to ensure a finite action set which means the models can be analysed through existing MDP reachability techniques). As for the standard digital clocks approximation, our approach does not support strict inequalities and comparisons between clocks. As we have action bigraphs, these clocks can live within the bigraph model itself, and a separate type of semantics is not needed (unlike for probabilistic bigraphs for example). We show our approach by an example, and the main idea is in Figure 4. We put all clocks into their own parallel region which we call the *clocks perspective*. A global clock is represented by an entity family $\mathsf{GC}(n)$—*i.e.* there is one entity for each $0 \leq n \leq t_{\max}$ for some max time $t_{\max}$—and is used to model *wall clock time*. Wall-time cannot be reset. Local clocks are entity families $\mathsf{LC}(n)$ where $0 \leq n \leq t_{\max}$ for each timed entity. For clarity of modelling, we place these in LocalClocks, although this is not strictly required. We identify a set of *timed* entities and expand their arity by 1 to link them to a specific local clock. Using this, we can *identify* a clock through the linked entity, *i.e.* clocks do not need specific identifiers of their own. Not

[1] Subject to any requirements about rule priorities and conditions.

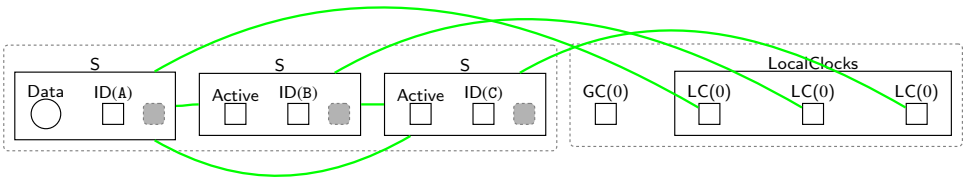


Figure 4: Example from Figure 1b extended with clocks perspective.

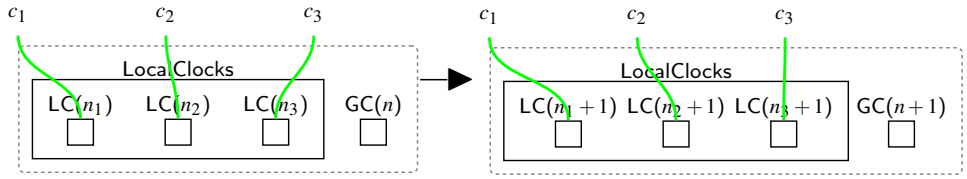


Figure 5: Reaction rule `clock_advance`(n_1,n_2,n_3,n) for a system with three timed entities.

all entities in the system will own a clock, *e.g.* Data is not timed. We assume all clocks are initialised to 0 in the initial state.

The approach to place clocks within their own region is a design choice, and because of the flexibility of bigraphs, other choices would be possible. For example, we could *nest* local clocks within particular timed entities instead of linking to them. We chose the extra region as it does not clutter any existing model, *i.e.* we can convert an existing model to a timed representation without significant changes (other than arity) in existing regions.

3.2 Reaction Rules for Digital Clocks

In our digital clocks approximation we have two main types of action: *discrete* actions which are user-defined and system specific, and *time* actions that deal with the passage of time. As actions in ABRSs are modelled as sets of reaction rules, the main challenge here is defining appropriate reactions for each type of action.

For timing actions, the main rule is `clock_advance`$(n_1,n_2,\ldots,n)$ shown in Figure 5 (for a system with three timed entities). This rule advances all local and global clocks simultaneously. All clocks advance at the same speed (one unit per application). This is a parameterised rule, which, like parameterised entities, defines a family of rules *i.e.* one for each possible value of the parameters. The *tick* action consists of the set of all possible `clock_advance`$(n_1,n_2,\ldots,n)$ rules for parameters drawn from $n,n_1,n_2,\ldots \leq t_{\max}$, for some max time $t_{\max}$. Due to the parameters only one instance of `clock_advance` will ever be enabled meaning this action always advances time with probability one.

For *discrete* actions we extend existing system rules whenever a time constraint must be met. We take rules that have a standard (untimed) definition of a bigraphs rule $(L \longrightarrow\!\!\!\!\blacktriangleright R)$ and, like with the clocks perspective, utilise parallel regions to link timed entities with their clocks as follows:

$$/c\,(\widehat{L}_c \parallel \mathsf{LC}(n)_c) \longrightarrow\!\!\!\!\blacktriangleright /c\,(\widehat{R}_c \parallel \mathsf{LC}(n')_c)$$

where $n' \in \{n,0\}$, notation $\widehat{\_}$ indicates the transformation over bigraphs described in Section 3.1 in which the entity type of timed entities has its arity increased by one, *i.e.* each entity has one more name. The

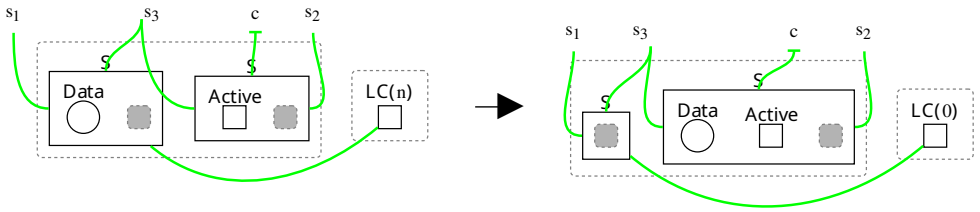


Figure 6: Reaction rule(s) `sending_data`(n). Time constraints are encoded by setting $2 \leq n \leq t_{\max}$.

subscript c indicates the name used to link a specific timed entity to a new clock placed in a *different* regions as specified by the $\|$ operator. Other entities are forbidden from linking over c as the link is *closed* (with $/c$). Importantly, this change also makes existing rules into parameterised rules (over parameter n). This is used later to encode timing constraints, *e.g.* we chose a set of parameters that define at what (local) *time* a particular rule can be applied. For rules that are already parameterised, we can simply add an additional parameter for the clock.

As an example we extend our rule in Figure 2, that allows sensors to send data, to only fire when the receiver is active, *and* at least two time units have passed *e.g.* $\mathsf{LC}(n) \geq 2$. As we are modelling an inequality we cannot do this with a single reaction rule and instead we provide a new parameterised rule `sending_data`(n), shown in Figure 6, that explicitly links the sending sensor to its local clock. The rule is only valid for $n \in \{2,3,\ldots,t_{\max}\}$ so this is the family of rules we generate. In this case, we include a clock reset on the right-hand-side of the rule, and clocks may only be reset during the application of a rule.

Note that the `clock_advance` rule has the same priority as the corresponding rules whose applications are triggered by clock constraints. This allows time to pass until the clock valuation satisfies the first invariant. In a such state, the non-deterministic choices are applicable *i.e.* the system can take a discrete action or permit the time to pass if it is not the last valid clock valuation in which an action must occur.

We must also encode any location-based clock invariants, *e.g.* locations that are only valid for $x \leq 2$. In practice this is used to force a transition to occur rather than allowing an indefinite wait within a particular location, and is almost always *true* (allowing indefinite waits) or a $\leq$ constraint. To encode these invaraints we make use of priority classes in the ABRS. For a state invariant, *e.g.* $t \leq x \leq n$, we add any outgoing transition rules $\mathtt{r}(n)$ such that $\{\mathtt{clock\_advance}(\ldots),\mathtt{r}(t),\mathtt{r}(t+1),\ldots,\mathtt{r}(\mathtt{n}-1)\} < \{\mathtt{r}(n)\}$. This means $\mathtt{r}(n)$ applies *before* any clock updates and *forces* a state transition before the invariant is broken.

4 Examples and Implementation

We implement our approach in BigraphER which also generates the MDPs (that are the semantics for our digital clocks representation). The exported MDPs can be formally checked using standard (probabilistic) model checkers, *e.g.* PRISM, allowing us to benefit from existing model checking algorithms.

Modelling a timed system with our approach follows these general steps:

- Define sets of clock valuations/invariants according to the system requirements.
- Define a new clock perspective and add clock entities equal to the number of the timed entities required, and one global clock. Increase the arity of timed entities by 1, and link to the local clock. All clocks are initialised to 0.

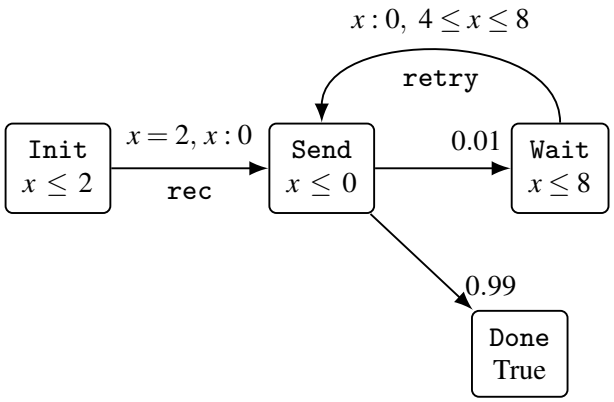


Figure 7: A probabilistic timed automata example model for a simple sending data process from [13].

- For a given $t_{\max}$ and number of timed entities, generate the set of rules in the form

$$\texttt{clock\_advance}(\{0,\ldots,t_{\max}\},\{0,\ldots,t_{\max}\},\ldots)$$

- Extend system rules by adding in the clock perspective when required to restrict their application based on the time constraints. Parameters for these rules must meet the clock invariants. Any state-based invariants are encoded using priorities.

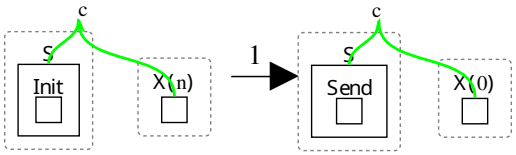
We illustrate our approach by encoding two different examples: a probabilistic timed automata example, and requests allocation in a cloud system [14]. The BigraphER models for both examples are in the appendix.

4.1 PTA example

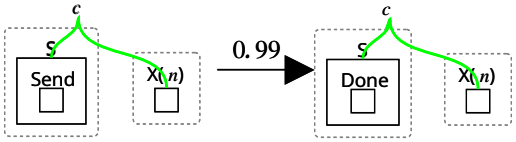
We apply our approach to the probabilistic timed automata example shown in Figure 7 and recreated from [13]. This simple communication system attempts to send data based on the constraints over clock x. Here we do not model the actual sends, only the state machine the system goes though. This is somewhat unnatural for bigraphs, where we are more concerned with global system updates (possibly of multiple agents) than encoding a particular state machine for an entity, but is used to illustrate that we can recover existing PTA semantics.

The system starts in the initial state `Init`. When the time has elapsed exactly 2 time units (as forced by the transition constraint *and* the clock invariant on the state) the system moves to the `Send` state. The clock invariant $x \leq 0$ forces data to be sent immediately. With probability 0.99 the system reaches its final state, `Done`, and with 0.01 probability it fails and moves to a `Wait` state. The system waits at least 4 time units and at most 8 time units (forced by the invaraints) before moving back to the `Send` state to retry. The clock is reset with each transition.

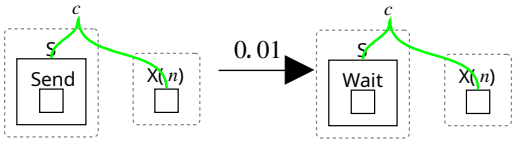
We start modelling this PTA example by defining the required time constraints. Here we use $\mathsf{X}(n)$ instead of $\mathsf{LC}(n)$ to be closer to the original example. For each state we define the valid clock valuations as follows. `init_transition` rule (Figure 8a) applies over $n \in \{0,1,2\}$ for `Init` state, and the rules that are associated to `Send` state (Figure 8b and Figure 8c) fire immediately upon receiving data *i.e.* at $n = 0$. While `wait_transition` rule (Figure 8d) is applicable over $n \in \{4,5,6,7,8\}$. For all parameters except 2 for `Init` and 8 for `Wait` states, the rules application should be in the same priority class as $\texttt{clock\_advance}(\ldots)$ rule allowing the non-deterministic behaviour: time passes or an data



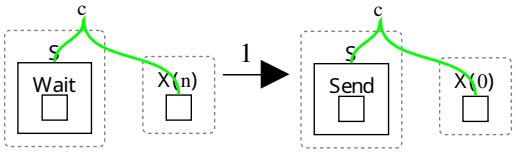
(a) Reaction rule `init_transition`(n), $n \in \{0,1,2\}$



(b) Reaction rule `send_transition_success`(n), $n = 0$



(c) Reaction rule `send_transition_fail`(n), $n = 0$



(d) Reaction rule `wait_transition`(n), $n \in \{4,5,6,7,8\}$

Figure 8: Reaction rules for the PTA example.

sends. However, as the invariants force leaving if more than $t_{\max}$ units elapse, `init_transition`(2) `send_transition`(0), and `wait_transition`(8) are in a higher priority class. We encode a single rule for each system transition, and show how the probabilities are encoded using rule weights[2], *e.g.* to allow `Send` state to move to either `Wait` or `Done`.

In Figure 9, we show the clocks bigraph model conforms to the probabilistic timed automata example by giving the transition system. For space, we only provide the first few transitions. Here the system moves from the initial state `Init` to the `Send` state. When the system is in the `Init` state and the clock constraints are satisfied, there are two possibilities: the time passes or an action occurs. Once the clock becomes $\mathsf{X}(2)$, the system **has to** move to `Send`.

4.2 Cloud System

We remodel the example presented in [14]. The authors also add time constructs to BRSs, but their work differs as it expresses clocks as nested entities and needs multiple reaction rules to advance clocks (which can cause unnecessary state-space explosion). They do not have a method to track approach wall-time and, as they do not use action BRSs, cannot model the (explicit action) non-deterministic behaviour of many real-time systems.

A cloud system is a collection of resources that include hardware, software, networks, etc. that is used to store, manage, and process data. That is end users (EU) send their requests (R) over the internet

[2]In this case, as there is only one agent moving state, the weights are equivalent to their probabilities since there will never be additional rule application matches to account for.

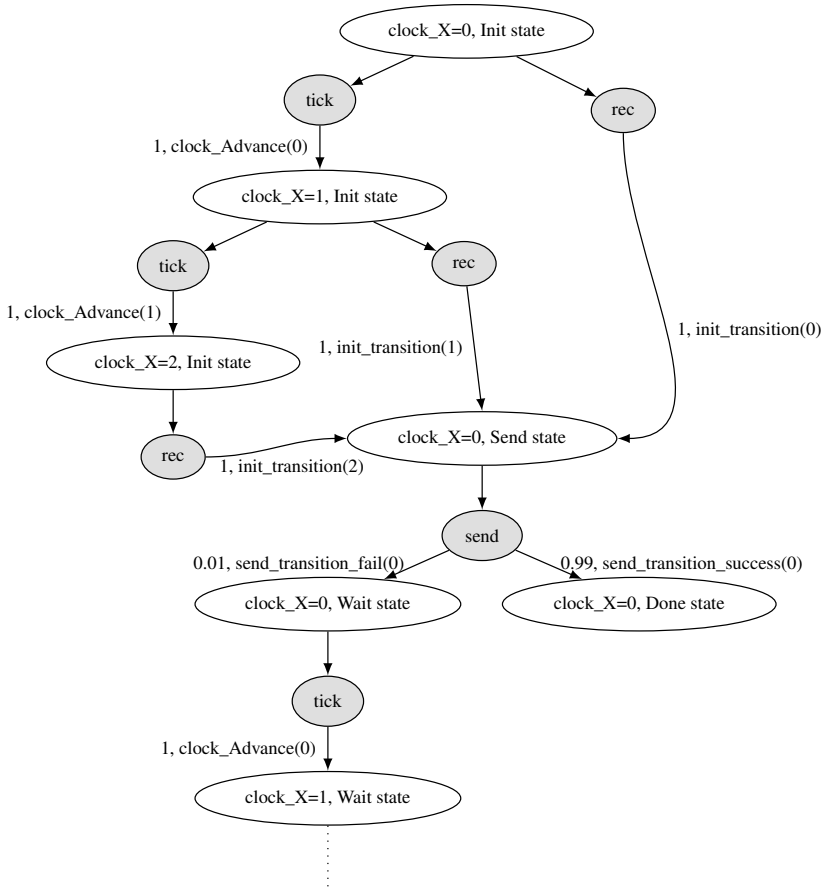


Figure 9: Resulting MDP (partial) for the probabilistic timed automata shown in Figure 7.

to virtual machines (VM) hosted on physical servers ($\mathsf{S}(i)$ with $i > 0$) within a data centre (DC) for processing. Standard bigraphs can model and analysis the structure and dynamic behaviour of cloud systems but not the time constraints that cloud applications usually have *e.g.* the duration of tasks. We consider a system that has two end users connected to each other via a link *e.g.* e_1; and two servers that are also connected via a link *e.g.* e_2. A data centre and an end user relate to each other via a link *e.g.* y_1. There are four requests each of them connects to its end user through a link *e.g.* x_1. The requests need to be processed using two servers. We model the same example as follows. We give each request a parameterised identifier ($\mathsf{ID}(r)$, where $r \in 1,2,3,4$). We also associate each request with different controls to reflect its current status. Initially all requests are in a (Wait) status; requests that are under processing have status Processing while status Result indicates that a request has been processed and returned to the end user. We define a clock for each request but, unlike in [14], we add a global clock that shows the wall-time. We also use a reaction rule that simultaneously advances all the clocks. We specify one reaction rule `sendingRequest` (Figure 10a) to send a request from an end user to a virtual machine for processing. Given that the servers have varying resources and thus require different amounts of time to process a request, two rules are applied to return the requests after processing, *i.e.* once a predefined time has elapsed. Rule `processRequest_S1` is for server $\mathsf{S}(1)$ that takes 2 time units and rule `processRequest_S2` is for server $\mathsf{S}(2)$ that needs 3 time units to process a request. We define the

same time constraint assumptions stated in [14] as integer sets. These sets are to be utilised with the timed reaction rules. That is, for each request there are four integer sets as follows. The *first one* is the valid clock valuations for a request. The *second* contains the time at which a request can be sent. The *last one* is two different sets that determine the time that a request should last according to the processing time for each server. We provide the four integer sets as follows:

- $N = \{n \mid 0 \leq n \leq t_{\max}\}$ for each request,
- $S = \{1,3,4,5\}$ for requests 1, 2, 3, and 4 respectively,
- $P_1 = \{s+2 \mid s \in S\}$ for requests processed on server $\mathsf{S}(1)$,
- $P_2 = \{s+3 \mid s \in S\}$ for requests processed on server $\mathsf{S}(2)$.

While in [14] virtual machine migrations are considered, here we allocate the process to a machine that is in Idle status indicating it is free and ready to receive a request. We model sending requests through reaction rule `sendingRequest` (Figure 10a). This rule allocates the request to an Idle server when a request's time constraint is satisfied. We then encode two rules that return requests back to the sender. Note that depending on the allocated server, only one of the return rules applies: rule `returnRequest_S1` takes the result back to EU for requests that are treated on server $\mathsf{S}(1)$ once 2 time units elapsed; while rule `returnRequest_S2` takes it back to EU after 3 time units passes for requests that are treated on server $\mathsf{S}(2)$. Figure 10b shows a return rule that applies when the time constraint (either p_1 or p_2) is met for $\mathsf{S}(i)$. We will show in the following section how our approach can be used to verify the interesting properties. We make the assumption that a request is always forced to send no more than 1 time unit *after* the deadline. This means processing time for delayed tasks can be faster than expected, *e.g.* $p_1 = s+2-1$. This could be accounted for using additional entities, *e.g.* Delayed, or out of time requests could be dropped. Bigraphs give the flexibility to experiment with these approaches with additional rules.

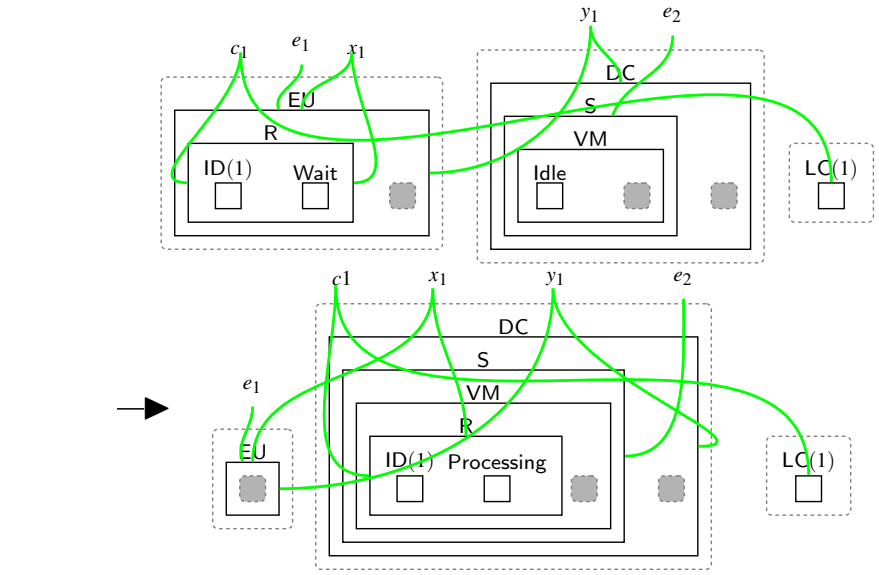
4.3 Model Analysis

To check the feasibility of our proposed modelling technique, we export the bigraph models into MDPs automatically using BigraphER. We can then verify them against required properties using PRISM by expressing properties in the Probabilistic Temporal Logic (PCTL) [9]. To help writing properties, we utilise *bigraph patterns* that allow states to be labelled based on matches [6], *e.g.* label any state that has a Data entity. Since clocks are just part of the model, we can also label clock values, *e.g.* $clock\_LC_0$ for $\mathsf{LC}(0)$. Interestingly, the clocks used in the properties are those in the model, *i.e.* they are just bigraph entities, rather than clock variables you would generate if you, for example, expressed the PTA directly in PRISM. This gives flexibility, *i.e.* we can write predicates over many different types of clock matches.

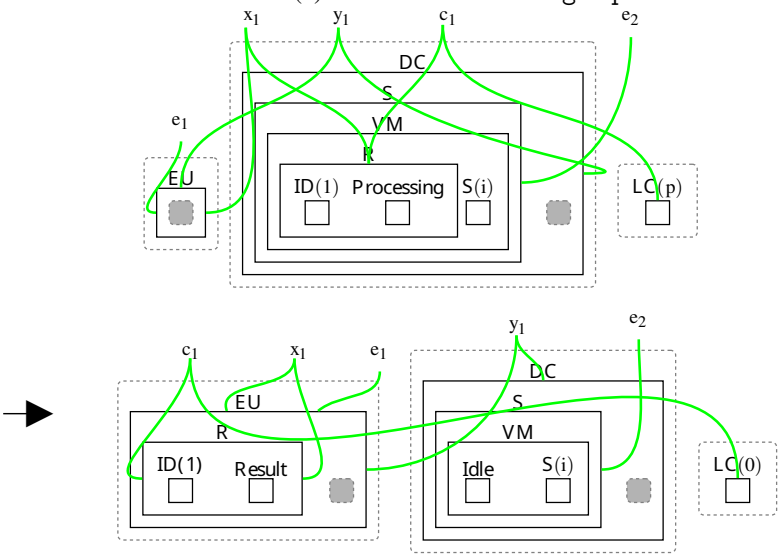
We make use of the following PCTL quantifiers to specify our properties: **A** (for all) and **E** (there exists), and path formulae, **F** (eventually), **X** (next), **U** (until) and $\wedge$ (and). We also use probability quantifier **P** to check the probability of reaching a specific state. For example, $\mathbf{P} \leq 0.5\,[\mathbf{F}\,stateA\,]$ means: (over all paths) is there at most a 0.5 probability that the system eventually reaches a state satisfying *stateA*, we label a state as *stateA* using a bigraph pattern.

We conduct model checking on several interesting properties. In the PTA example (Section 4.1), we can perform probabilistic reachability properties such as "Is there at least a 0.99 percent probability that the system moves to `Done` state successfully", which holds for this system.

$$\mathbf{P} \geq 0.99\,[\mathbf{F}\,(\texttt{In\_Done\_state})\,] \tag{1}$$



(a) Reaction rule `sendingRequest`.



(b) Reaction rule `returnRequest`(p,i), $p = 3$ when $i = 1$ and $p = 4$ when $i = 2$.

Figure 10: Processing rules for the first request.

We can also utilise clock predicates to check properties relating to time *e.g.* "eventually there is at least a 99% chance that the system moves to the `Done` state and resets clock X", which is true for our system. Importantly, these are based on our clock entities, not time-bounded properties within PRISM which operate on their own time units.

$$\mathbf{P} \geq 0.99\,[\,\mathbf{F}\,(\,\texttt{In\_Done\_state} \wedge \texttt{clock\_X}_0)\,] \tag{2}$$

We can check clock invariants are satisfied, *e.g.* that the system is not be in `Wait` state after 8 time units

have elapsed:

$$\mathbf{A}\left[\neg\left(\mathbf{F}\left(\texttt{In\_Wait\_state}\right)\wedge\left(\texttt{clock\_X}_9\right)\right)\right] \tag{3}$$

As expected this is true in all cases.

For the cloud system we can, for example, check the first request is always sent once *one* time unit has elapsed. As we use next (**X**), the request must be sent immediately before any other state transition.

$$\mathbf{A}\left[\mathbf{F}\ \texttt{clock}_1\wedge\left(\mathbf{X}\ \texttt{request}_1\texttt{\_sent\_to\_S}\right)\right] \tag{4}$$

5 Related work

Many modelling formalisms have been extended to real-time systems by introducing clocks. Timed CCS [25] extends Milner's CCS by introducing the time notion to create concurrency models for real-time systems. Timed CCS introduces a variable to record time delays *e.g.* before a message arrives. Similar to our work, timed CCS considers the positive real number including zero as the time domain for convenience, but the model can deal with another numerical domain *e.g.* the natural numbers. Timed Petri nets [26] extends Petri nets to allow time triggered transitions. It uses tokens to allow transitions to start and then they are placed into the output when the firing process terminates. The authors use random variables with continuous or discrete probability distribution functions for the firing time. Timed π-Calculus [18] extends the π-Calculus with continuous time to describe and reason about concurrent Cyber-Physical Systems and real-time systems. An executable operational semantics of π-Calculus is developed in Logic Programming to model concurrency.

Graph Transformation Systems (GTS) has been extended into Probabilistic Timed Graph Transformation Systems (PTGTSs) which enable modelling and analysing structure dynamic and timed and probabilistic behaviour of embedded systems [15]. The clock is a typed node that is contained in a graph to identify the nodes that are used for time measurement only. In this work, PTGTSs is formally mapped into Probabilistic Timed Automata (PTA) where the Probabilistic Timed Structure (PTS) of the mapped PTGTSs is equal to the PTS of the resulting PTA, hence they both satisfy the same set of PTCTL properties. The obtained PTA can be checked using PRISM. However, the mapping process does not consider three aspects of the PTA. First, since the PTA does not consider valuations in the labelling function, constraints are ignored in the mapping process so the constraint should be true for any such atomic proposition. It also considers the PTGTS that does not show timelocks during its execution instead of finding and removing them when mapping PTGTS into PTA. Finally, the GTS state space that is constructed for PTGTS is up to isomorphism as preserved clock nodes are respected which may affect the state space finiteness of the resulting PTA. In our work, we bound the clock valuations which ensures finite transition systems. Additionally, in case that a system frequently resets its local clocks, the employment of the global clock guarantees the finiteness of the transition system. To prevent timelocks and so obtain a proper transition between reachable states, we assign the maximum states invariant valuation to the rules that are in a higher priority.

Bigraphs have been applied to model the location and connectivity of components of structure-aware mobile systems using BigrTimo that combines the rTiMO process algebra and bigraphs [24]. It uses real-time constraints to control actions by showing the waiting time for communication. The work in [14] explicitly encodes clocks as entities within bigraphs to model and reason about cloud applications, and shares some similarities with our approach as discussed in Section 4.2. It adds a set of clocks and a set of clocks constraints that are associated with nodes. It then utilises two different types of rules: (1) a set of

reaction rules to advance all clocks, all clocks are advanced at the same speed; (2) instantaneous rules[3] that are executed only when there is a match and time constraints of one or many nodes are satisfied. These rules can also update the clock constraints and resets the clocks. However, when a new time constraint is satisfied, the time cannot elapse and another instantaneous rule may apply subsequently. In contrast, we encode timed aspects as action bigraphs resulting in an MDP transition system that explicitly models the non-deterministic behaviour of timed systems, that then can be formally checked by different model checking tools. The work uses Real-Time Maude language [7] and its TCTL model checker to implement and analyse the approach. State-space explosion is reduced here by employing a strategy that models all clocks as a separate region allowing us to use only one reaction rule to advance all clocks simultaneously. We imitate the wall-time by utilising the global clock entity.

6 Conclusion

Bigraphical Reactive Systems (BRSs) have been successfully used to model a wide range of systems but, until now, they had no explicit support for real-time systems. We overcome this limitation by introducing a modelling technique that uses the digital clocks approximation of (probabilistic) timed automata to encode timing aspects. This approach relies on Action Bigraphs, which have as a semantics Markov Decision Processes (MDPs), meaning we can express both probabilistic and timing behaviour within models. Using BigraphER, we encode the proposed strategy and verify the MDP corresponding to each model using PCTL model checking. Using two examples, we show our approach supports multiple clocks and the transition system obtained via rewriting faithfully encodes the digital-clock approximation of the behaviour of a real-time system.

Our approach suffers from the same limitations as the digital-clock approximation *i.e.* currently we do not support diagonal clocks and strict inequalities. We mitigate state space explosion by adopting the following two strategies. First, we bound clock valuations thus we obtain finite transition systems. Second, we allow the base time unit to be specified in each model, effectively allowing clocks to advance by multiple ticks in one step. Another limitation of our approach is that we assume clocks are always synchronised, *i.e.* they all progress at the same speed, which might not always be the case in scenarios like wireless sensor networks and IoT.

In future, we will develop syntactic support for clock constraints in the BigraphER language to generate real-time ABRS models like the ones considered in the paper. Sorting schemes [5] (type systems for bigraphs) will ensure correct separation of clocks and model entities. Syntax and sorts will ensure our extended set of reaction rules is correct by construction. We also aim to extend our approach to also support diagonal clocks and strict inequalities.

Acknowledgement

This work is supported by the UK EPSRC projects CHEDDAR (EP/X040518/1) and CHEDDAR Uplift (EP/Y037421/1), and an Amazon Research Award on Automated Reasoning.

[3] Silent rules that do not appear in an output transition system.

References

[1] Rajeev Alur & David L. Dill (1994): *A Theory of Timed Automata*. *Theor. Comput. Sci.* 126(2), pp. 183–235, doi:10.1016/0304-3975(94)90010-8.

[2] Blair Archibald, Muffy Calder & Michele Sevegnani (2020): *Conditional Bigraphs*. In Fabio Gadducci & Timo Kehrer, editors: *Graph Transformation - 13th International Conference, ICGT 2020, Held as Part of STAF 2020, Bergen, Norway, June 25-26, 2020, Proceedings*, *Lecture Notes in Computer Science* 12150, Springer, pp. 3–19, doi:10.1007/978-3-030-51372-6_1.

[3] Blair Archibald, Muffy Calder & Michele Sevegnani (2022): *Probabilistic Bigraphs*. *Form. Asp. Comput.* 34(2), doi:10.1145/3545180.

[4] Blair Archibald, Muffy Calder, Michele Sevegnani & Mengwei Xu (2022): *Modelling and verifying BDI agents with bigraphs*. *Sci. Comput. Program.* 215, p. 102760, doi:10.1016/J.SCICO.2021.102760.

[5] Blair Archibald & Michele Sevegnani (2024): *A Bigraphs Paper of Sorts*. In Russ Harmer & Jens Kosiol, editors: *Graph Transformation - 17th International Conference, ICGT 2024, Held as Part of STAF 2024, Enschede, The Netherlands, July 10-11, 2024, Proceedings*, *Lecture Notes in Computer Science* 14774, Springer, pp. 21–38, doi:10.1007/978-3-031-64285-2_2.

[6] Steve Benford, Muffy Calder, Tom Rodden & Michele Sevegnani (2016): *On Lions, Impala, and Bigraphs: Modelling Interactions in Physical/Virtual Spaces*. *ACM Trans. Comput. Hum. Interact.* 23(2), pp. 9:1–9:56, doi:10.1145/2882784.

[7] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer & Carolyn L. Talcott, editors (2007): *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. *Lecture Notes in Computer Science* 4350, Springer, doi:10.1007/978-3-540-71999-1.

[8] Davide Grohmann & Marino Miculan (2007): *Directed Bigraphs*. In Marcelo Fiore, editor: *Proceedings of the 23rd Conference on the Mathematical Foundations of Programming Semantics, MFPS, LA, USA, April 11-14, 2007*, *Electronic Notes in Theoretical Computer Science* 173, Elsevier, pp. 121–137, doi:10.1016/J.ENTCS.2007.02.031.

[9] Hans Hansson & Bengt Jonsson (1994): *A Logic for Reasoning about Time and Reliability*. *Formal Aspects Comput.* 6(5), pp. 512–535, doi:10.1007/BF01211866.

[10] Thomas A. Henzinger, Zohar Manna & Amir Pnueli (1992): *What Good Are Digital Clocks?* In Werner Kuich, editor: *Automata, Languages and Programming, 19th International Colloquium, ICALP92, Vienna, Austria, July 13-17, 1992, Proceedings*, *Lecture Notes in Computer Science* 623, Springer, pp. 545–558, doi:10.1007/3-540-55719-9_103.

[11] Jean Krivine, Robin Milner & Angelo Troina (2008): *Stochastic Bigraphs*. In Andrej Bauer & Michael W. Mislove, editors: *Proceedings of the 24th Conference on the Mathematical Foundations of Programming Semantics, MFPS, PA, USA, May 22-25, 2008*, *Electronic Notes in Theoretical Computer Science* 218, Elsevier, pp. 73–96, doi:10.1016/J.ENTCS.2008.10.006.

[12] Marta Z. Kwiatkowska, Gethin Norman & David Parker (2011): *PRISM 4.0: Verification of Probabilistic Real-Time Systems*. In Ganesh Gopalakrishnan & Shaz Qadeer, editors: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, *Lecture Notes in Computer Science* 6806, Springer, pp. 585–591, doi:10.1007/978-3-642-22110-1_47.

[13] Marta Z. Kwiatkowska, Gethin Norman, David Parker & Jeremy Sproston (2006): *Performance analysis of probabilistic timed automata using digital clocks*. *Formal Methods Syst. Des.* 29(1), pp. 33–78, doi:10.1007/S10703-006-0005-2.

[14] Fateh Latreche & Faiza Belala (2019): *Timed CTL checking of time critical cloud applications using timed bigraphs*. *Int. J. Crit. Comput. Based Syst.* 9(4), pp. 379–406, doi:10.1504/IJCCBS.2019.106818.

[15] Maria Maximova, Holger Giese & Christian Krause (2018): *Probabilistic timed graph transformation systems*. *J. Log. Algebraic Methods Program.* 101, pp. 110–131, doi:10.1016/J.JLAMP.2018.09.003.

[16] Robin Milner (2009): *The Space and Motion of Communicating Agents*. Cambridge University Press, doi:10.1017/CBO9780511626661.

[17] Martin L. Puterman (1994): *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics, Wiley, doi:10.1002/9780470316887.

[18] Neda Saeedloei & Gopal Gupta (2013): *Timed π-Calculus*. In Martín Abadi & Alberto Lluch-Lafuente, editors: *Trustworthy Global Computing - 8th International Symposium, TGC Argentina, August 30-31, 2013, Revised Selected Papers*, *Lecture Notes in Computer Science* 8358, Springer, pp. 119–135, doi:10.1007/978-3-319-05119-2_8.

[19] Hamza Sahli, Faiza Belala & Chafia Bouanaka (2015): *A BRS-Based Approach to Model and Verify Cloud Systems Elasticity*. In Keith G. Jeffery & Dimosthenis Kyriazis, editors: *1st International Conference on Cloud Forward: From Distributed to Complete Computing, October 6-8, 2015, Italy*, *Procedia Computer Science* 68, Elsevier, pp. 29–41, doi:10.1016/J.PROCS.2015.09.221.

[20] Hamza Sahli, Thomas Ledoux & Éric Rutten (2019): *Modeling Self-adaptive Fog Systems Using Bigraphs*. In Javier Cámara & Martin Steffen, editors: *Software Engineering and Formal Methods - SEFM 2019 Collocated Workshops: CoSim-CPS, ASYDE, CIFMA, and FOCLASA, Norway, September 16-20, Revised Selected Papers*, *Lecture Notes in Computer Science* 12226, Springer, pp. 252–268, doi:10.1007/978-3-030-57506-9_19.

[21] Michele Sevegnani & Muffy Calder (2015): *Bigraphs with sharing*. *Theor. Comput. Sci.* 577, pp. 43–73, doi:10.1016/J.TCS.2015.02.011.

[22] Michele Sevegnani & Muffy Calder (2016): *BigraphER: Rewriting and Analysis Engine for Bigraphs*. In Swarat Chaudhuri & Azadeh Farzan, editors: *Computer Aided Verification - 28th International Conference, CAV, ON, Canada, July 17-23, 2016, Proceedings, Part II*, *Lecture Notes in Computer Science* 9780, Springer, pp. 494–501, doi:10.1007/978-3-319-41540-6_27.

[23] Michele Sevegnani, Milan Kabác, Muffy Calder & Julie A. McCann (2018): *Modelling and Verification of Large-Scale Sensor Network Infrastructures*. In: *23rd International Conference on Engineering of Complex Computer Systems, ICECCS, Australia, December 12-14, 2018*, IEEE Computer Society, pp. 71–81, doi:10.1109/ICECCS2018.2018.00016.

[24] Wanling Xie, Huibiao Zhu & Qiwen Xu (2017): *BigrTiMo-A Process Algebra for Structure-Aware Mobile Systems*. In: *22nd International Conference on Engineering of Complex Computer Systems, ICECCS, Fukuoka, Japan, November 5-8, 2017*, IEEE Computer Society, pp. 50–59, doi:10.1109/ICECCS.2017.13.

[25] Wang Yi (1991): *CCS + Time = An Interleaving Model for Real Time Systems*. In Javier Leach Albert, Burkhard Monien & Mario Rodríguez-Artalejo, editors: *Automata, Languages and Programming, 18th International Colloquium, ICALP91, Spain, July 8-12, 1991, Proceedings*, *Lecture Notes in Computer Science* 510, Springer, pp. 217–228, doi:10.1007/3-540-54233-7_136.

[26] Wlodek M Zuberek (1991): *Timed Petri nets definitions, properties, and applications*. *Microelectronics Reliability* 31(4), pp. 627–644, doi:10.1016/0026-2714(91)90007-T.

Appendix A BigraphER implementation of the examples in Section 4

```
################  PTA Example #################

atomic fun ctrl X(n) = 1 ;
ctrl S = 1;
atomic ctrl Init = 0;
atomic ctrl Send = 0;
atomic ctrl Wait = 0;
atomic ctrl Done = 0;

################  Reaction Rules #################

##Init
fun react init_transition(n) =
  S{c}.Init || X(n){c}
  -[1]->
  S{c}.Send || X(0){c};

## Send
fun react send_transition_success(n) =
  S{c}.Send || X(n){c}
  -[0.99]->
  S{c}.Done || X(n){c};

fun react send_transition_fail(n) =
  S{c}.Send || X(n){c}
  -[0.01]->
  S{c}.Wait || X(n){c};


# Wait
fun react wait_transition(n) =
  S{c}.Wait || X(n){c}
  -[1]->
  S{c}.Send || X(0){c};

#Done
react done_done =
  S{c}.Done -[1]-> S{c}.Done;

################ Clock Advance Rule #################
fun react clock_advance(n) =
  X(n){c}
  -[1]->
  X(n + 1){c};

################ Predicates #################
fun big clock_X(n) = X(n){c};
big in_Init_state = S{c}.Init;
big in_Send_state = S{c}.Send;
big in_Wait_state = S{c}.Wait;
big in_Done_state = S{c}.Done;

################ Initial State #################
big example_PTA = /c (S{c}.Init || X(0){c});

 begin abrs
  int n = {0,1,2,3,4,5,6,7,8};
  int maxInitT = {2};
  int init_Sending_Time = {0,1};
  int maxSendT = 0;
  int maxWaitT = 8;
  int wait_Sending_Time={4,5,6,7};
```

```
63
64   init example_PTA;
65
66   rules = [
67     # Higher in the list => higher priority
68     {done_done,
69      init_transition(maxInitT),
70      send_transition_fail(maxSendT),
71      send_transition_success(maxSendT),
72      wait_transition(maxWaitT)},
73
74     {clock_advance(n),
75      wait_transition(wait_Sending_Time),
76      init_transition(init_Sending_Time)}
77   ];
78
79   actions = [
80     send={send_transition_success,send_transition_fail},
81     retry={wait_transition},
82     rec={init_transition},
83     deadlock={done_done},
84     tick={clock_advance}
85   ];
86
87   preds = {
88     in_Init_state,
89     in_Send_state,
90     in_Wait_state,
91     in_Done_state,
92     clock_X(n)
93   };
94 end
```

```
################  Cloud System Example #################

ctrl FrontEnd = 0;
ctrl EU = 3;
ctrl R = 2;
atomic ctrl Processing = 0;
atomic ctrl Result = 0;
atomic ctrl Wait = 0;
atomic ctrl Idle = 0 ;
atomic ctrl Stop = 0 ;
ctrl BackEnd = 0;
ctrl DC = 1;
ctrl S = 1;
atomic ctrl S1 = 0;
atomic ctrl S2 = 0;
ctrl VM = 0;
atomic fun ctrl ID(i) = 0;
ctrl LocalClock = 0;
atomic fun ctrl LC(requestClock) = 1;
atomic fun ctrl GC(globalClock) = 0;

################# Reaction Rules #################
fun react clock_advance(request1Clock, request2Clock, request3Clock, request4Clock, gc)
    =
  LocalClock.( LC(request1Clock){loclock1} | LC(request2Clock){loclock2} | LC(
      request3Clock){loclock3} | LC(request4Clock){loclock4} ) | GC(gc)
  -[1]->
  LocalClock.( LC(request1Clock +1 ){loclock1} | LC(request2Clock +1 ){loclock2} | LC(
      request3Clock +1 ){loclock3} | LC(request4Clock +1 ){loclock4} ) | GC(gc+1) if !
      Stop in ctx;

fun react sendingRequest(i, r1_Send_Time) =
  EU{x1,y1,e1}.(R{x1,c1}.(ID(i) | Wait ) | id ) ||  DC{y1}.( S{e2}.VM.(Idle | id ) | id
       )  || LC(r1_Send_Time){c1}
  -[1]->
  EU{x1,y1,e1}.id  ||  DC{y1}.( S{e2}.VM.(R{x1,c1}.(ID(i)  | Processing) | id ) | id  )
       || LC(r1_Send_Time){c1} ;

fun react returnRequest_S1(i, s1_Process_Time) =
  EU{x1,y1,e1}.id  ||  DC{y1}.( S{e2}.VM.(R{x1,c1}.(ID(i)  | Processing) | S1 ) | id  )
       || LC(s1_Process_Time){c1}
  -[1]->
  EU{x1,y1,e1}.( R{x1,c1}.( ID(i) | Result ) | id ) ||  DC{y1}.( S{e2}.VM.(Idle | S1 )
      | id ) || LC(0){c1};

fun react returnRequest_S2(i, s2_Process_Time) =
  EU{x1,y1,e1}.id ||  DC{y1}.( S{e2}.VM.(R{x1,c1}.(ID(i)  | Processing) | S2 ) | id  )
      || LC(s2_Process_Time){c1}
  -[1]->
  EU{x1,y1,e1}.( R{x1,c1}.( ID(i) | Result ) | id ) ||  DC{y1}.( S{e2}.VM.(Idle | S2 )
      | id )  || LC(0){c1};

react done =
  EU{x1,x2,e1}.( R{x1,c1}.(Result | id ) | R{x2,c2}.(Result | id ) ) | EU{y1,y2,e1}.( R
      {y1,c3}.( Result | id ) | R{y2,c4}.(Result | id ) )
  -[1]->
  EU{x1,x2,e1}.( R{x1,c1}.(Stop | id ) | R{x2,c2}.(Stop | id ) ) | EU{y1,y2,e1}.( R{y1,
      c3}.( Stop | id ) | R{y2,c4}.(Stop | id ) ) ;


################# Initial State #################
big cloudSystem =
/x1/x2/y1/y2/e1/e2/c1/c2/c3/c4 (
 FrontEnd.(
```

```
   EU{x1,x2,e1}.( R{x1,c1}.(Wait | ID(1) ) | R{x2,c2}.(Wait | ID(2) ) ) | EU{y1,y2,e1
       }.( R{y1,c3}.( Wait | ID(3) ) | R{y2,c4}.(Wait | ID(4) ) )
)
|| BackEnd.(DC{e1}.( S{e2}.VM.(Idle | S1 ) | S{e2}.VM.(Idle | S2 )) )
|| LocalClock.( LC(0){c1} | LC(0){c2} |  LC(0){c3} | LC(0){c4} ) | GC(0)
);

################## Predicates #################
fun big request_Sent_to_S1_at(i, x) =
   S{e2}.VM.(R{x1,c1}.(ID(i) | Processing ) | S1 )
|| LC(x){c1};

fun big request_Return_at(i, x) =
R{x1,c1}.(ID(i) | Result ) || LC(x){c1};


begin abrs
  int i = {1, 2, 3, 4};
  int request1Clock={0,1,2,3,4,5,6,7,8};
  int request2Clock={0,1,2,3,4,5,6,7,8};
  int request3Clock={0,1,2,3,4,5,6,7,8};
  int request4Clock={0,1,2,3,4,5,6,7,8};
  int gc={0,1,2,3,4,5,6,7,8,9,10,11,12};

  int r1_Send_Time={1};
  int r1_Max_Send_Time={2};
  int r2_Send_Time={3};
  int r2_Max_Send_Time={4};
  int r3_Send_Time={4};
  int r3_Max_Send_Time={5};
  int r4_Send_Time={5};
  int r4_Max_Send_Time={6};

  int r1_Process_Time_S1={3};
  int r1_Process_Time_S2={4};
  int r2_Process_Time_S1={5};
  int r2_Process_Time_S2={6};
  int r3_Process_Time_S1={6};
  int r3_Process_Time_S2={7};
  int r4_Process_Time_S1={7};
  int r4_Process_Time_S2={8};

  int x ={0,1,2,3,4,5,6,7,8,9,10,11};


init cloudSystem;

  rules = [
        {done},
        {returnRequest_S1(1, r1_Process_Time_S1), returnRequest_S1(2,
            r2_Process_Time_S1), returnRequest_S1(3, r3_Process_Time_S1),
            returnRequest_S1(4, r4_Process_Time_S1),

        returnRequest_S2(1, r1_Process_Time_S2), returnRequest_S2(2, r2_Process_Time_S2
            ), returnRequest_S2(3, r3_Process_Time_S2), returnRequest_S2(4,
            r4_Process_Time_S2)},

        {sendingRequest(1,r1_Max_Send_Time), sendingRequest(2,r2_Max_Send_Time),
            sendingRequest(3,r3_Max_Send_Time), sendingRequest(4, r4_Max_Send_Time)} ,

        {sendingRequest(1,r1_Send_Time), sendingRequest(2,r2_Send_Time), sendingRequest
            (3,r3_Send_Time),
        sendingRequest(4, r4_Send_Time), clock_advance(request1Clock,request2Clock,
            request3Clock, request4Clock, gc)}
   ];
```

```
110
111 actions=[
112  send={sendingRequest},
113  return = { returnRequest_S1, returnRequest_S2 },
114  tick = {clock_advance},
115  stop = {done}
116 ];
117
118 preds = {
119 request_Sent_to_S1_at(i, x), request_Return_at(i, x)};
120 end
```

Pedagogy of Teaching Pointers in the C Programming Language using Graph Transformations

Adwoa Donyina
Computer Science Department
Tagliatela College of Engineering
University of New Haven
Connecticut, USA
adonyina@newhaven.edu

Reiko Heckel
School of Computing and Mathematical Sciences
University of Leicester
Leicester, UK
rh122@leicester.ac.uk

Visual learners think in pictures rather than words and learn best when they utilize representations based on graphs, tables, charts, maps, colors and diagrams. We propose a new pedagogy for teaching pointers in the C programming language using graph transformation systems to visually simulate pointer manipulation. In an Introduction to C course, the topic of pointers is often the most difficult one for students to understand; therefore, we experiment with graph-based representations of dynamic pointer structures to reinforce the learning. Groove, a graph transformation tool, is used to illustrate the behaviour of pointers through modelling and simulation. A study is presented to evaluate the effectiveness of the approach. This paper will also provide a comparison to other teaching methods in this area.

1 Introduction

Every student has a distinct learning style, which should be enabled by their learning environment to allow them to succeed. In [14] the authors evaluate different learning styles in a move towards more student-centered classes. There are visual and non-visual learners; however visual learners tend to gain higher comprehension [12]. Within the realm of visual teaching techniques there are different depths, from using visual aids such as PowerPoint slides via static more or less formal representations such as pictures, maps, diagrams and charts, to dynamic and interactive visualisations.

We are proposing a visual learning pedagogy using graph transformations for the modelling and simulation of pointer manipulations in C. This is often the most difficult topic for students to understand in an introductory C programming course. We report on an experiment in utilising a graph-based representation of dynamic pointer structures during a revision session at the end of the course and compare the test results of this class to those of an otherwise equivalent class a year earlier.

This paper is organized as follows. In Section 2 we discuss background on the traditional methods of teaching C pointers at university and briefly introduce concepts needed to understand our proposed pedagogy. In Section 3 we present our approach of representing pointer manipulations using graph transformation. In Section 4 we illustrate it by a simulation based on an example. In Section 5 we present results from an experiment using the approach during the revision stage of a programming course. In Section 6 we compare and contrast related work. In Section 7 we will conclude the paper and discuss future work.

J. Endrullis, D. Grzelak, T. Heindel and J. Kosiol (Eds):
GCM 2023 and GCM 2024
EPTCS 417, 2025, pp. 117–133, doi:10.4204/EPTCS.417.7

2 Background

In many universities, C is the first programming language on the curriculum. The Computer Science and Cyber Security programme at the University of New Haven (UNH) teaches computer languages in the order of C, C++, Python, Java. In more advanced classes students utilize these languages to study data structures and algorithms, and more specific subjects. The initial programming course teaches many fundamental skills to help students starting their programming career. Learning to program is difficult, and pointer manipulation poses particular challenges to students [15]. Therefore, there is a need to explore new pedagogical approaches to support teaching the hardest concepts in C programming.

The Introduction to C Programming class at UNH is based on Chapters 1-13 of the textbook [5]. Pointers are introduced at the end of the term, to ensure that students understand the basic topics prior to being introduced to this machine-oriented way of thinking.

1. Computers and Systems
2. Programs and Programming
3. Fundamental Concepts
4. Objects, Types and Expressions
5. Using Functions and Libraries
6. More Repetition and Decision
7. Using Numeric Types
8. Trouble with Numbers
9. Program Design
10. An Introduction to Arrays
11. Character Data and Enumerations
12. An Introduction to Pointers
13. Strings

For many years pointer manipulation was taught only using the textbook [5] as a resource. This uses informal illustrations of pointer structures and their changes during program execution but naturally they are restricted to a few examples and do not allow to simulate alternatives based on different input states or to show the effects of erroneous code. To improve the teaching of pointer manipulation by visual means we propose to use graph transformation [4, 7], in particular Groove [11], as a pedagogical aid.

Graph transformation is a *rule-based* approach which represents procedural knowledge in terms of a set of “IF-THEN" rules. These rules define the preconditions and effects of the basic activities. This form of modelling is in contrast to the *control-oriented* approach, which focuses on the ordering of events.

Formally, graph transformation rules consist of left hand sides, right hand sides and negative application conditions (NACs), all of which are different graphs; however Groove combines them in one graph with colour coding [11]. The tool also supports advanced concepts such as graph constraints, negative application conditions, multi objects and patterns, and control flow specifications as well as the analysis by model checking.

3 Pointer Manipulation using Graph Transformation

In addition to the traditional means of teaching pointers in the C programming language, we are proposing a new pedagogy using graph transformation systems to visually simulate pointer manipulation. The Groove [11] graph transformation tool is used to create and simulate executable pointer models.

As part of a our experiment, we designed a type graph to model basic C pointer structures as shown in Figure 1. This type graph represents how C pointers relate to addresses and objects, such as array, int or char, that may reside at those addresses. Addresses represent memory cells linked by *succ* edges if they are consecutive. Each pointer may refer to an address and each address may contain an object. An

array is a consecutive sequence of addresses containing objects, and the Array node has a *fst* edge to the pointer referencing the first address of the array. The array's length is specified using attribute *len*. We use the Object type's name attribute to refer to a variable in the program code, but this is information is never used in the rules describing the operations and serves only to improve readability of the generated graphs. In particular, neither addresses nor pointers directly represent variables. Our type graph does not account for more advanced concepts such as objects of different size or the distinction between heap and stack memory, which were not covered in our introductory module.

An important design decision is the explicit modelling of addresses representing memory cells and their consecutive organisation in memory (by means of *succ* edges). Their status as free or allocated is recorded by the Boolean attribute *free* to reflect the effect of C memory functions `malloc()` and `free()`. This is a lower level of representation than the obvious model of pointer structures where pointers are edges between objects; however, this implementation-oriented model also allows us to capture operations such as dereferencing `*` and address-of `&`. As importantly, we can show situations where the structure violates referential integrity, e.g., when a pointer refers to an address which is either free or holds no object, potentially leading to inconsistencies we would like to illustrate in the execution of an erroneous program. We define graph constraints over the type graph to represent both referential integrity and basic constraints on the model. For the latter:

- *fst* edges have *-to-1 cardinality;
- *ref* edges have *-to-0..1 cardinality, where a pointer with no *ref* edge has value *null*;
- *cont* edges have 0..1-to-0..1 cardinality, i.e., while each object in memory has a unique address we allow partial representations where the address of, for example, a pointer object, is not relevant, and an address can only be the first address of at most one object;
- *succ* edges have 0..1-to-0..1 cardinality, i.e., each address apart from the first / last has a predecessor / successor; specifically, *succ* edges should form a single chain representing a linear address space.

These constraints must always be satisfied for an instance graph to be *well-formed*, i.e., model a pointer structure. They can be formally specified using graph constraints such as those in Figure 2a (for all Array nodes there exists an outgoing *fst* edge to a Pointer node) and Figure 2b (there is an Array node with two outgoing *fst* edges to different Pointer nodes). Groove [11] supports quantified graph elements using forall (∀) and exist (∃) nodes. Every node referring to these by an @ edge is correspondingly quantified, while edges are assigned implicitly to the quantifier of their source or target, whichever is higher, with nesting of quantifiers shown by *in* edges. Hence, the constraint in Figure 2a expresses the first-order formula $\forall$ *a:Array* $\exists$ *p:Pointer, f:fst.* $src(f) = a \wedge tar(f) = p$. The constraint in Figure 2b uses a pattern with a negated equality edge indicating that the two Pointer nodes must be matched to different nodes in the graph. Then, the formula *G (isWFfstEx & ! notWFfstToV)* defines an invariant for the cardinality of *fst* edges as stated in the first condition above where *G* is the temporal logic operator stating that the formula holds in all states. Analogously, we defined the other three cardinalities.

In order to represent a structure that is *correct*, i.e., respects referential integrity, we require that a *ref* edge never points to an address that is free (not allocated) nor has no *cont* edge (does not contain data).

Graph constraints for these two conditions are shown in Figures 2c and 2d respectively, jointly invoked by the formula *G (! notRIrefTofree & ! notRIrefWOcont)*. The second constraint uses a negative condition (in red) to indicate the absence of an object at the address. This captures the case where a pointer refers to an address that is not in use, so accessing data from this location leads to a logical error.

Well-formedness should be invariant while correctness is a property of a graph that may be violated by simulations modelling the execution of erroneous programs, which is pedagogically valuable. Either formula can be validated using Groove's model checker.

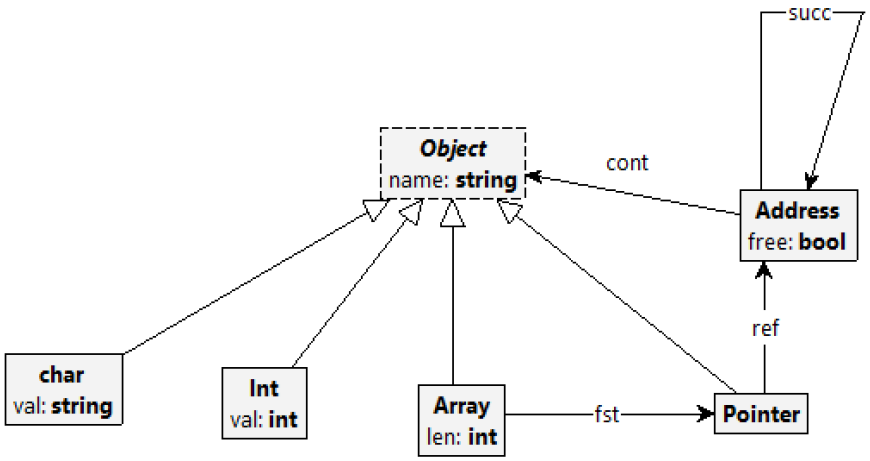
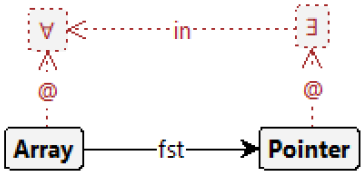
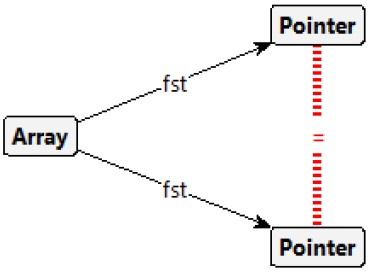


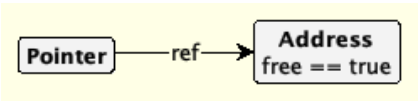
Figure 1: Type graph for pointer structures



(a) isWFfstEx, using nested universal and existential quantifiers



(b) notWFfstToV, using red inequality constraint edge labelled =



(c) notRIrefTofree

(d) notRIrefWOcont

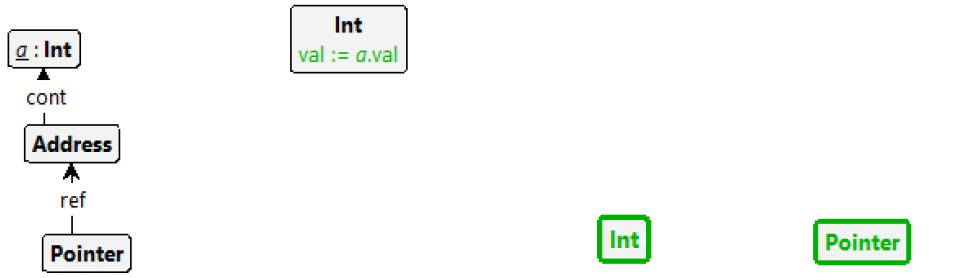
Figure 2: Graph constraints for well-formedness (WF) and referential integrity (RI)

A number of graph transformation rules encode basic operations of pointer manipulations. The rules use the Groove notation where left- and right-hand sides are integrated into one graph, with colours representing which elements are required and preserved (solid black outline), deleted (dashed blue outline), and created (bolder green outline). Forbidden elements (red dotted outline) represent negative application conditions.

- The rule in Figure 3a copies the value of the `int` referred to by a pointer into another `int` object. For instance, if `int *pt` and `int s`, the rule realises `s = *pt`.
- The rule in Figure 3b creates an `int` object.
- The rule in Figure 3c creates a `null` pointer.
- The rule in Figure 4a assigns to a pointer the address of another pointer. For instance, if `int *pt` and `int *pt2` are both not `null`, the rule realises `pt2=pt`.

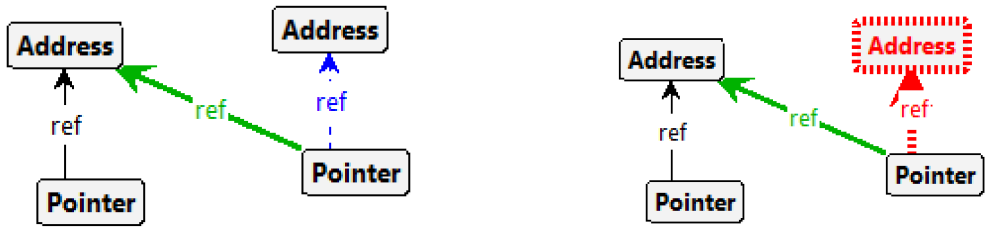
- The rule in Figure 4b is similar to the one in Figure 4a, however it assigns the address to a `null` pointer. For instance, if `int *pt` is not `null` but `int *pt2` is `null`, the rule realises `pt2=pt`. This is expressed as a negative application condition, shown in red dotted outline, specifying the absence of a *ref* edge to an Address node.
- The rule in Figure 5a assigns to a pointer a different address.
- The rule in Figure 5b assigns to a pointer the address of (the first element of) an existing array. For instance, if `int a[]={4,7,9}` and `int *pt` is `null`, the rule realises `pt=a` and `pt=a[0]`. Similar rules are defined to access other positions in the array using the *succ* relation.
- The rule in Figure 6a assigns to a pointer the address of an existing `int` object. After executing this rule, the pointer refers to the `int`'s address.
- The rule in Figure 6b assigns to a `null` pointer an address which now contains an existing int object. For instance, if `int b=25` and `int *pt` is `null`, the rule realises `*pt=b`.

This set of rules is enough to illustrate the scenario in Section 4 but not complete for the entire range of pointer operations in the C language.



(a) Copy value to int object (b) New int (c) New pointer

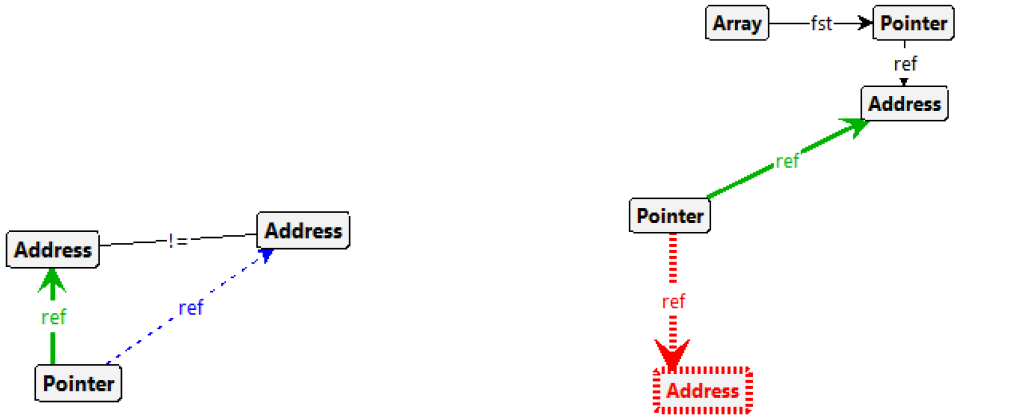
Figure 3: Basic rules with int & pointers



(a) Not null pointer assigned address of another (b) Null Pointer assigned address of another

Figure 4: A pointer is assigned the address of another pointer

In C code, pointer operations are denoted by the `&` and `*` symbols. The asterisk `*`, apart from being used for pointer declaration, returns the object at the address referenced by a pointer. In terms of our model this means to navigate from the Pointer to the Object via the Address node along *ref* and *cont*. Instead, `&` returns the address of an object, navigating *cont* in the reverse. Hence, if `int *pt` defines a pointer to the address of `int t=30` (or, more compactly: `int t=30; int *pt=&t`), then



(a) Pointer is assigned new address (b) Null Pointer is assigned address of array

Figure 5: Pointer is assigned address of array

(a) Int is stored at (free) address referred to by pointer (b) Null pointer is assigned address where int is stored

Figure 6: Int is stored at address and referred to by pointer

- `pt` returns the address referenced by the pointer, or null if the pointer is undefined;
- `&pt` returns the address of the pointer as an object;
- `*pt` returns the int value 30;
- `pt` and `&*pt` both return the address of the int.

Hence, if `int s` and `int t` are defined then `s = *pt` assigns the value of the int referred to by pointer `pt` to an int variable `t`. This use of `&` and `*` is illustrated in Figure 7. The dotted circle annotations shows what the notation is referring to, i.e., the pointer's content address and int value, respectively.

The result of executing the following C code is represented by the instance graph in Figure 8.

```
int s = 0;
int t = 0;
int age[] = { 30, 65, 41, 23 };
int *agep, *maxp;
```

In the bottom left are objects of type int represented by the variables `s` and `t`, currently unconnected.

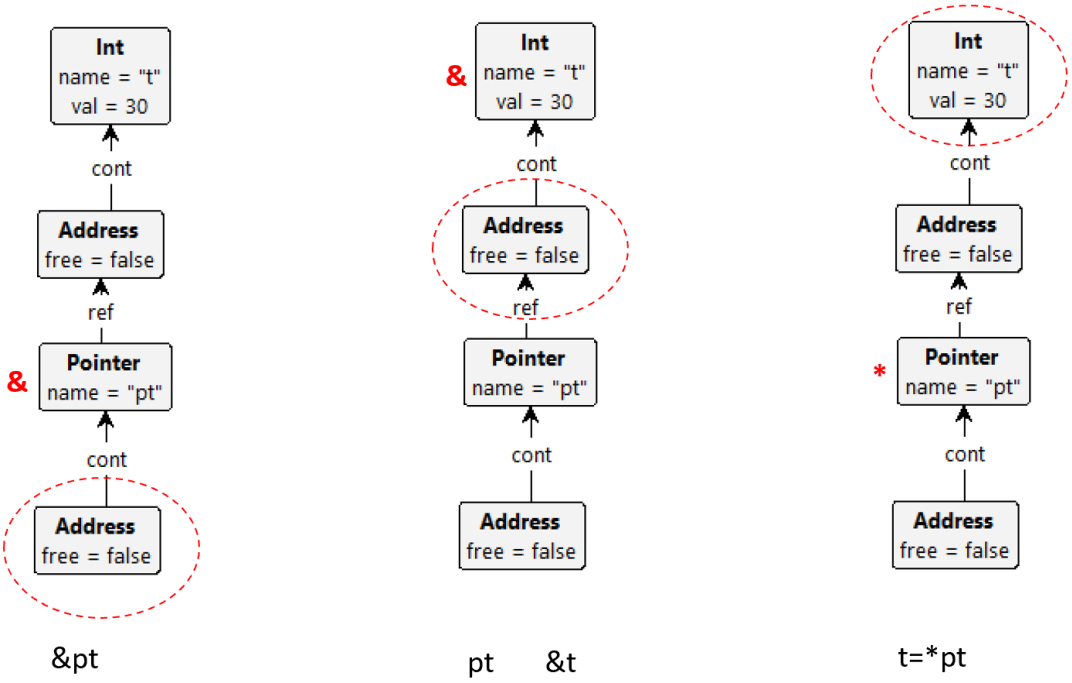


Figure 7: Effect of & vs * Notation

The age array refers to the pointer holding the address of the first element of the array. This and the three subsequent addresses contain the corresponding int values of the initial array.

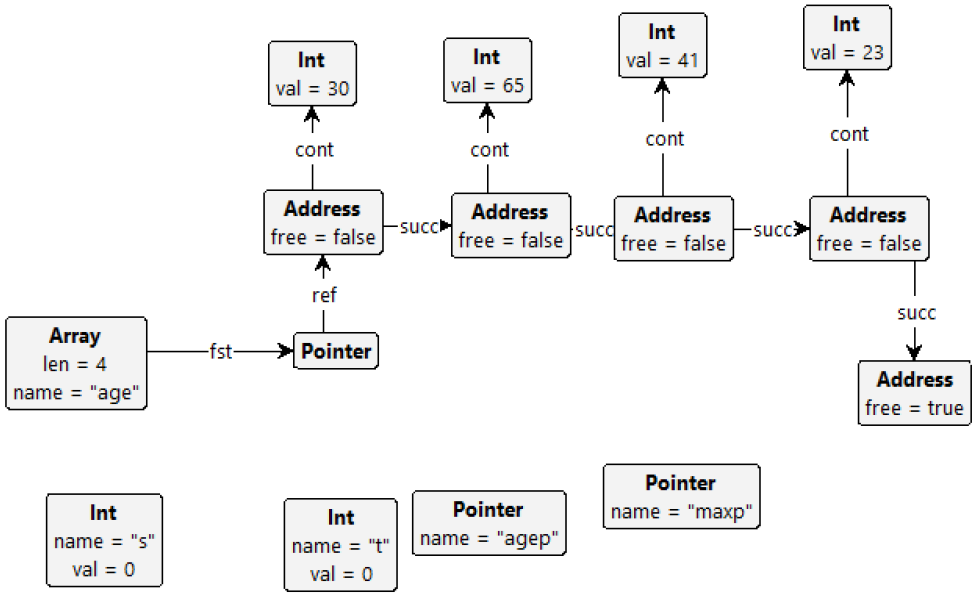


Figure 8: Instance graph representing start graph of simulation

4 Simulation

The instance graph defined in Figure 8 is the start graph for our simulations, whose runs were utilized to illustrate the following code.

```
s=*age; OR  s=age[0];
agep=age;
agep= &age[3];
*maxp=t;
```

A sample simulation using some of the GT rules defined in Section 3 is shown in Figures 9- 16.

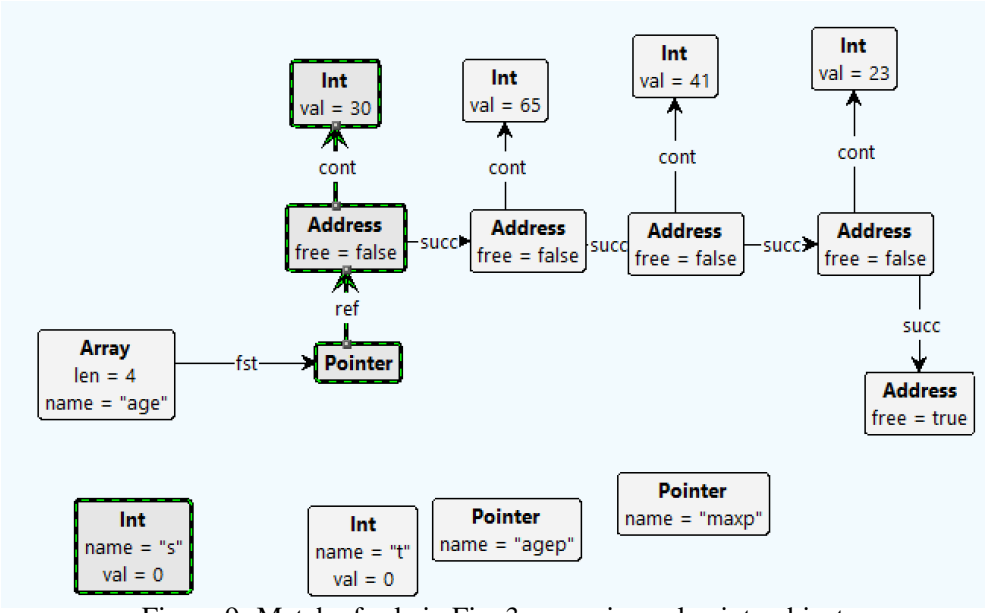


Figure 9: Match of rule in Fig. 3a: copying value into object

5 Evaluation

The pedagogical impact of using graphical simulations for teaching C pointer manipulation was tested on a class of students on the Introduction to C Programming course in Spring 2023. As a benchmark, we used another class of students on the same course during Spring 2022. Both sets of students were taught pointers from the same textbook [5] and given the same review questions.

The visualisations that the Spring 2023 class received included PowerPoint slides summarising the lecture on pointers, an in-class simulation demo, practice exercise and a Zoom recording of the lecture. The PowerPoint slides introduced the students to the type graph, start graph, reviewed basic pointer notation, and illustrated how the pointer notation is represented in the graph, as shown in Figure 7. After reviewing basic concepts, the Groove Simulator tool was used to further explore and understand the start graph, which matched a homework exercise problem (Table 1). A simulation run was then performed by illustrating different possible operations specified by rules. Once this was complete, we went back to the original homework question by adding the correct notation to the code on the whiteboard. Both Spring classes were assessed using the same questions on their test and final exam, similar to the following textbook question [5].

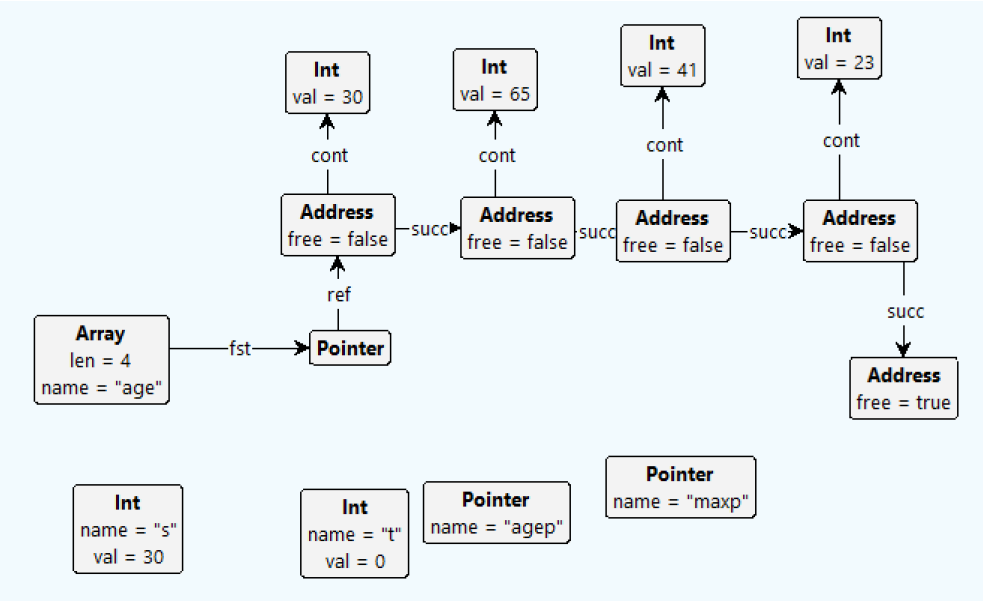


Figure 10: Rule in Fig. 3a applied for `s=*age;`

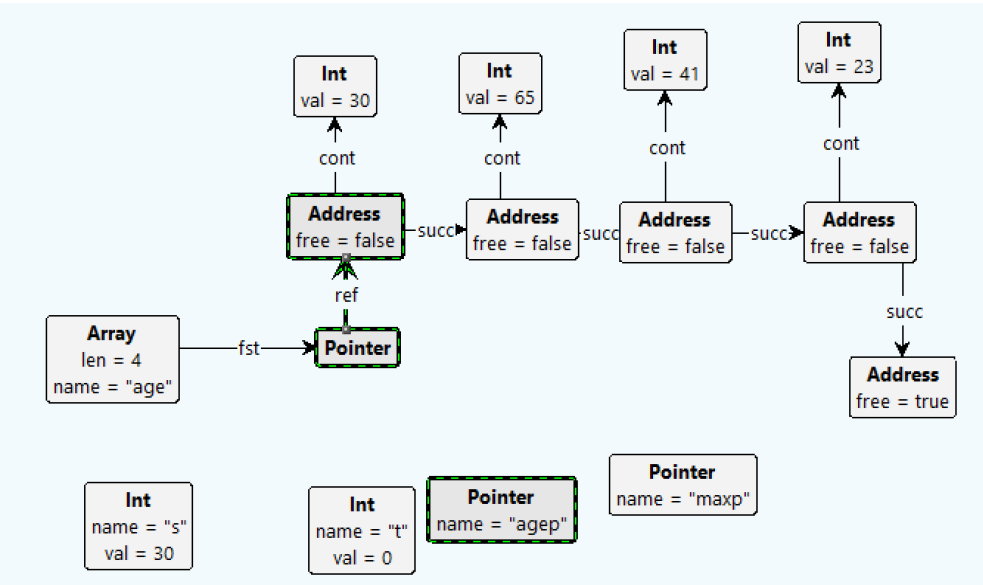


Figure 11: Match of rule in Fig. 4b: assigning a null pointer to another pointer's referent

Question: Complete each of the following C statements by adding an *asterisk* `*`, *ampersand* `&`, or *subscript* `[ ]` wherever needed to make the statement do the job described by the comment. Exercise question and answers are shown in Table 1. These question require a clear understanding of pointers and the notations used to operate on them. Use these declarations:

```
int s=0;
int t=0;
int age[] =  30, 65, 41, 23 ;
int * agep, * maxp;
```

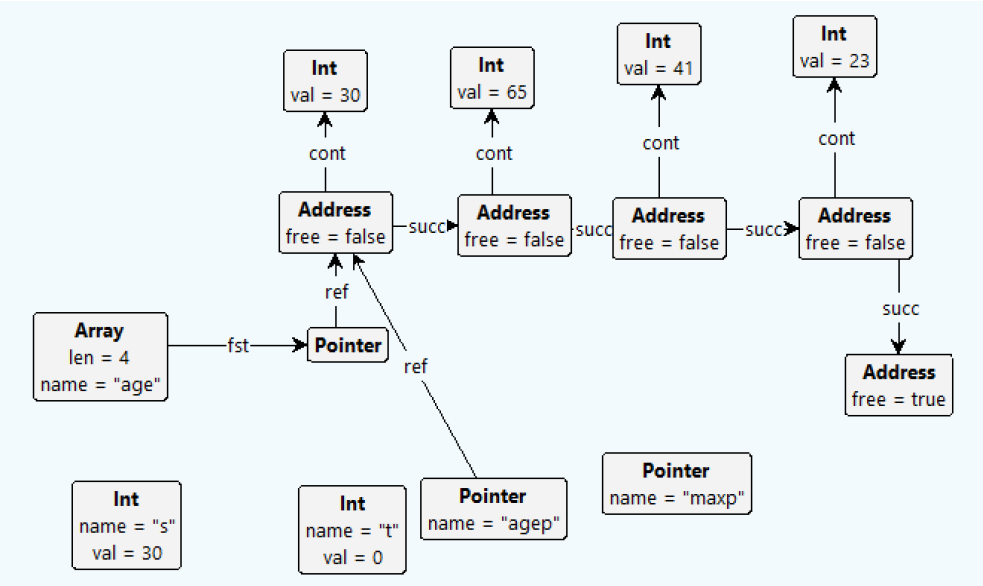


Figure 12: Rule in Fig. 4b applied for `agep=age;`

Question: Add an `*`,`&`, or *subscript `[ ]`* wherever needed	Answer
a. // Make agep refer to first age in array `agep = age;`	a. `agep = age;`
b. // Copy value of agep's referent into s. `s = agep;`	b. `s = *agep;`
c. // Copy 65 into agep's referent. `agep = age[];`	c. `*agep = age[1];`
d. // Make maxp refer to agep's referent. `maxp = agep;`	d. `maxp = agep;`
e. // Store mean of 2nd and last ages in agep's referent. `agep = (age[]+age[])/2;`	e. `*agep = (age[1]+age[3])/2;`
f. // Read into third array slot. `scanf( "%hi", age[] );`	f. `scanf( "%i", &age[2] );`
g. // Read into agep's referent. `scanf( "%hi", agep );`	g. `scanf( "%i", agep );`
h. // Print agep's referent. `printf( "%hi", agep );`	h. `printf( "%i", *agep );`

Table 1: Exercise: Question & Answer

5.1 Results

We compare the test results of the three classes (Spring 2022, 2023 & Fall 2022) and present some observations gathered from a survey of the Spring 2023 class. A value-added score [10] was used to

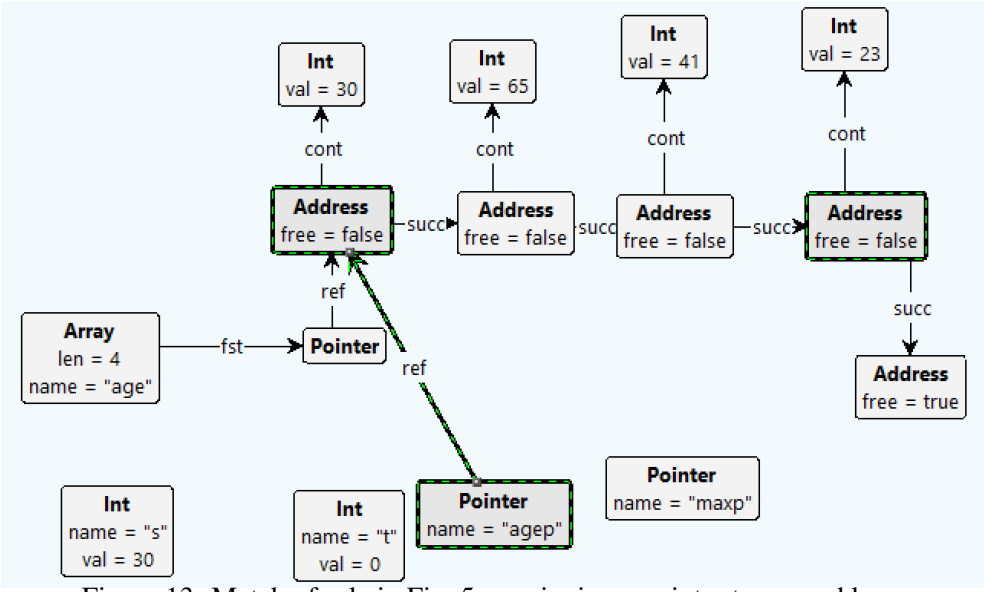


Figure 13: Match of rule in Fig. 5a: assigning a pointer to new address

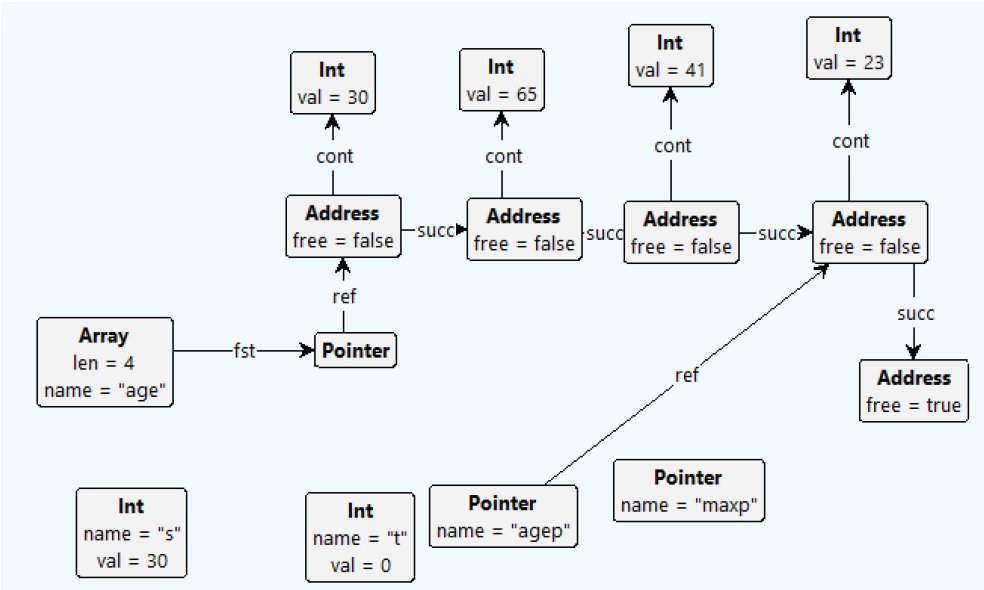


Figure 14: Rule in Fig. 5a applied for `agep= &age[3];`

compare the test results of the three classes.

A detailed evaluation was done comparing Spring 2023, Spring 2022, Fall 2022 overall value added scores and pointer test scores. In each semester, each student's value added score was calculated based the expected test score from their previous two test in comparison to their actual test 3 mark. The individual students' value added scores were than averaged in each semester. Results are as follows:

- Spring 2022: -17.76833333
- Fall 2022: -4.1818
- Spring 2023: -12.4559375

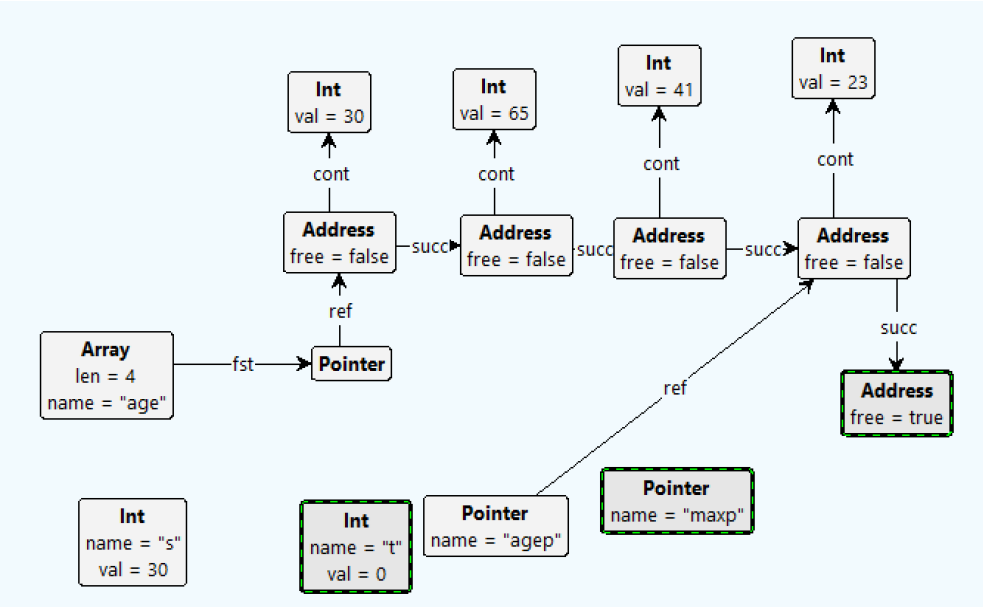


Figure 15: Match of rule in Fig. 6b: null pointer is assigned to the address of a int

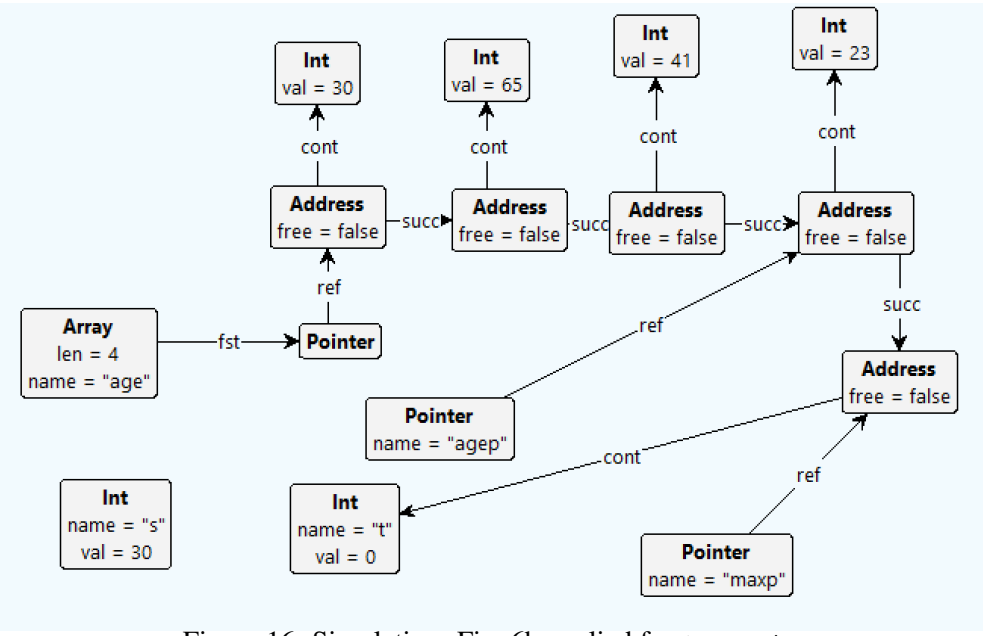


Figure 16: Simulation: Fig. 6b applied for `*maxp=t;`

The value added measures (VAM) [13] was determined based on the performance of the students in the various tests. First, the expected Test 3 scores were calculated for each individual student based on their Test 1 and 2 grades. Then the expected Test 3 score was compared to the actual score which then formed the basis of the VAM. In all cases, the VAM scores were negative because Test 3 was harder material then the prior two tests, which therefore produced overly optimistic predictions. Figure 17 shows the value added measures, the pointer test scores for Test 3 and the final exam. The results show that amounts of the negative value added measures are decreasing from Spring 2022 to 2023; hence

the effectiveness of the pedagogy is demonstrated. The validity of this statistical analysis is limited by the fact that students in different cohorts had different aptitudes. Spring semester student tend to be either retaking the C programming course, or they take the course later because they initially entered the university with poor math scores. However, the value-added scores shows that the pedagogy introduced in Spring 2023 positively impacted test and final exam pointer scores.

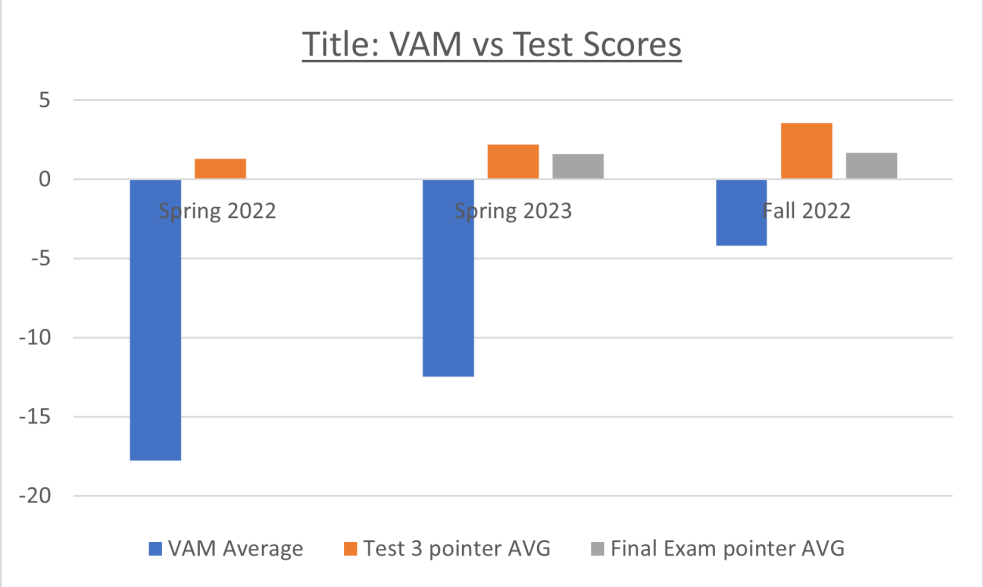


Figure 17: Evaluation Result: Value Added Measure vs Pointers test results in 3 semesters

Figure 18 shows the survey results for the question: "The visual representation helped me to understand what happens when the code is executed". Students were then asked to comment on the open-ended

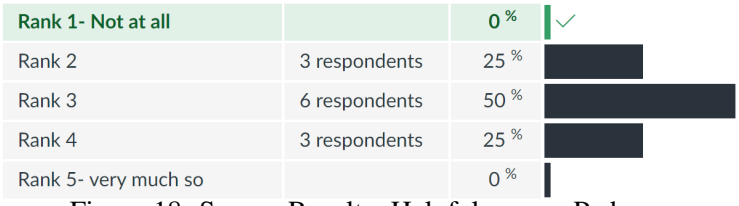


Figure 18: Survey Results: Helpfulness on Pedagogy

question: “How did it help, what was the additional insight gained from the graphical view?” Some of the anonymous responses were as follows:

- "Although it is still a slightly confusing topic for me, I realize that when trying to understand pointers it is very important to understand where everything is being stored and where things are being directed. The graphical view helped me visualize this easier. "
- "It helped me better understand that pointers are useful in programs and the graphical view helped me visualize the flow of pointers and how they operate in a program."
- "It was difficult to visualize how pointers worked and having a graphical view helped me understand how it pointed to different memory locations."
- "I’m more of a visual learner, so it helped me understand it in more than one way."
- "It helped me understand how connected it could be with the tree with all the connecting parts"

6 Related Work

While there are other novel approaches to teaching C pointers, we believe that our pedagogy provides a distinctive angle. [3] presents a traditional approach to teaching pointer manipulation used at a vocational college in China, which they believe to be an effective method. The approach distinguishes the key concepts of a memory address as name, address and content. They also used pictorial representations, and small coding examples. In addition to the traditional approach they also presented pointers in terms of analogies. For instance, a teacher office building contains room with room numbers (addresses) containing teachers (content). Otherwise [3] it is similar to the traditional method used at UNH as discussed in Section 2. However, authors agree that many students have difficulties in understanding pointers using the classical learning approach [16] and that there is a need for novel solutions.

[9] took on a Value Trace Problem (VTP) approach on pointers, where they provide source code, a set of questions and hints to a group of students. The questions ask the students to specify the value of a variable or output message from the code. The results are validated using string matching. Their approach was very similar to the traditional teaching with added hints and study questions, not visual but focused on actual memory address values. For example, given code `char test1[] = 'o','k','a','y','a','m','a';` and hint: *The address of `test1[0]` is 0028FF30*, what is the address of `'test1[1]'` [9]. This level of detail is avoided in our approach, where addresses are abstract nodes whose numerical representation is hidden. This comes at the cost of *succ* edges to represent their linear order, but supports the same concepts in relation to pointers, addresses and their operations `* ,&`.

[15] focuses on approaches to teach C pointers to mechanical engineering students, using a hydraulic press as an example. They addressed multi-dimensional arrays with pointers to read/write values from the memory of embedded devices. The main similarity here is that the method is also visual; however the visualisation was a depiction of a physical object. The key difference is that their their pedagogy is domain-specific, intended for mechanical engineering students while we make no such restriction.

Another approach to teaching programming uses serious games. The Perobo serious game is inspired by the operation of computer memory [16] using a constructivist approach to encourage the learner to complete activities such as completing definitions (on variables), guessing types for pointer declarations, and matching Random Access Memory (RAM). Our approach is implementation-oriented and has a simulation component. Both approaches are visual, however in different ways and different levels of abstraction. Our visualisation is at a lower level of abstraction, focusing on object, address, and pointer relations shown as graphs. Instead, [16] presents a visual game based on real-world metaphors. For instance, one of the game activities has a matrix depicting different memory locations, containing different types of robots. One of the robots represents a pointer while the others represent variables. The user is required to associate the pointer with the right address based on the values of the variables.

Graphs and graph rewriting have also been used for the *verification* of pointer programs. *Shape analysis* aims to check correctness with respect to constraints demanding, for example, that a data structure has a tree or list shape. Representations are often at a higher level of abstraction, where objects are nodes and pointers are edges, rather than the implementation-oriented model in our approach. For example, [6] propose a notion of shapes defined by a context-free graph grammar and an algorithm for static shape analysis of graph transformers. They also introduce a notation for shape types and transformers in C. [2] and [1] implement shape types by graph reduction to define and validate structures such as cyclic lists, linked lists, binary search trees, red-black trees and balanced binary tree. They have an algorithm for shape safety of operations such as search, insertion and deletion. [8] use graph grammars to obtain finite abstractions of pointer-manipulation programs to find bugs due to dereferencing of null pointers and memory leaks. Neither of these approaches aims at the pedagogy of teaching pointer manipulations.

In conclusion, our approach is more abstract than using numerical pointer values while being generic rather than depicting real-world objects and scenarios, using simulations to animate graphs representing pointer structures. Other approaches that also represent pointer manipulation by graphs transformation have different motivations and are presented at different level of abstractions.

7 Conclusion and Future Work

We presented an initial experiment in graph transformation for teaching pointer manipulation in C and reported on our experience with a class in the Spring term of 2023. The results are promising but not conclusive because, although we accounted for prior attainment using a value added score, a range of other factors could have affected students' performance across two years.

Our approach was re-evaluated using a more general model during the Fall 2023 semester in two C programming classes of 25 students each. Both classes were taught pointers using our new pedagogy and we compared the results to the Fall 2022 classes, which were not taught using the approach. The Fall 2023 students also had hands-on exposure to the Groove tool as independent extra credit project. We used similar pointer questions in their test and final exam to evaluate them, but due to a different model the results are incomparable to the ones presented here.

In particular, we updated the type graph in order to represent the linear address space by numerical address attributes rather than successor edges. This makes for a simpler representation, in line with how addresses work at machine level. A visualisation of pointers as edges can be derived from the numerical representation.

The Fall 2023 course began at the end of August 2023, however the syllabus was adjusted to ensure there will be adequate time to review pointers using the updated approach and introduce the students to graph transformation concepts and the Groove tool. Hence, Chapter 1 and Chapter 2: Programs and Programming of [5] were completed in the first week of the class (rather than starting in the second week as in Spring 2023); therefore by, November the students should have completed the majority of the required chapters outlined in Section 2.

In the Spring 2023 evaluation discussed in Section 5, the approach was only used to revise pointer, and was demonstrated to the students. This Fall the approach is used in the original delivery of the material. Also, students were be provided with a small tutorial on graph transformation and Groove in order to prepare them to have interactive experiments using the tool and the updated C pointer graph transformation system. A full report on this experiment is beyond the scope of this paper because it would require the introduction of a new model and different experiemntal setup.

Further developments will include:

- Defining a complete set of rules for pointer manipulation in C, including memory management using `malloc()`, `calloc()`, `realloc()` and `free()` and validating these using a range of sample programs.
- Considering function calls, including the difference between call by reference and call by value, and distinguishing stack from heap memory.
- Adding referential integrity constraints to the curriculum to help explain what are desirable (consistent) pointer structures, possible inconsistencies, and how they may affect program behaviour.
- Adding subject-specific visualisations for arrays and consecutive addresses to align notation to what may be found in textbooks.

Acknowledgment: We would like to thank the students of the Introduction to C Programming class at the University of New Haven, CT, USA for their participation.

References

[1] Adam Bakewell, Detlef Plump & Colin Runciman (2004): *Checking the Shape Safety of Pointer Manipulations*. In Rudolf Berghammer, Bernhard Möller & Georg Struth, editors: *Relational and Kleene-Algebraic Methods in Computer Science*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 48–61, doi:10.1007/978-3-540-24771-5_5.

[2] Adam Bakewell, Detlef Plump & Colin Runciman (2004): *Specifying Pointer Structures by Graph Reduction*. In John L. Pfaltz, Manfred Nagl & Boris Böhlen, editors: *Applications of Graph Transformations with Industrial Relevance*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 30–44, doi:10.1007/978-3-540-25959-6_3.

[3] Liping Deng, Xianfang Zou & Wenjuan Hu (2016/12): *Pointer Teaching of C Language in Higher Vocational Colleges*. In: *Proceedings of the 2016 International Conference on Advances in Management, Arts and Humanities Science*, Atlantis Press, pp. 438–441, doi:10.2991/amahs-16.2016.93.

[4] H. Ehrig, K. Ehrig, U. Prange & G. Taentzer (2006): *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[5] Alice E. Fischer, David Eggert & Stephen M. Ross" (2000): *Applied C An Introduction and More*. McGraw-Hill, Boston, MA.

[6] Pascal Fradet & Daniel Le Métayer (1997): *Shape Types*. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, Association for Computing Machinery, New York, NY, USA, p. 27–39, doi:10.1145/263699.263706.

[7] Reiko Heckel & Gabriele Taentzer (2020): *Graph Transformation for Software Engineers*. Springer International Publishing, doi:10.1007/978-3-030-43916-3. Available at `http://graph-transformation-for-software-engineers.org/`.

[8] Jonathan Heinen, Christina Jansen, Joost-Pieter Katoen & Thomas Noll (2015): *Verifying pointer programs using graph grammars*. *Science of Computer Programming* 97, pp. 157–162, doi:10.1016/j.scico.2013.11.012. Special Issue on New Ideas and Emerging Results in Understanding Software.

[9] Xiqin Lu, Nobuo Funabiki, Htoo Htoo Sandi Kyaw, Shune Lae Aung & Nem Khan Dim (2021): *An Improvement of Value Trace Problems for Pointer Study in C Programming*. In: *2021 IEEE 3rd Global Conference on Life Sciences and Technologies (LifeTech)*, pp. 148–152, doi:10.1109/LifeTech52111.2021.9391784.

[10] Jack Mountjoy & Brent Hickman (2021): *The Returns to College(S): Relative Value-Added and Match Effects in Higher Education*. *SSRN Electronic Journal*, doi:10.2139/ssrn.3926957.

[11] Arend Rensink (2023): *Groove-GRaphs for Object-Oriented VErification*. `http://aiweb.techfak.uni-bielefeld.de/content/bworld-robot-control-software/`. [Online; accessed 20-May-2023].

[12] Marvin Brent Roark (1998): *Different Learning Styles: Visual vs. Non-Visual Learners Mean Raw Scores in the Vocabulary, Comprehension, Mathematical Computation, and Mathematical Concepts.* Available at `https://eric.ed.gov/?id=ED430774`.

[13] Brevard Public Schools (2015): *Value added measure*. Available at `https://www.youtube.com/watch?v=0m8J9sve6V4`.

[14] Pawan Asghar Talib (2022): *The Responses and Attitudes of the University of Nottingham Students toward Learning Styles.* *Arab World English Journal* 13(1), pp. 394 – 407. Available at `https://files.eric.ed.gov/fulltext/EJ1336023.pdf`.

[15] B. Vogel-Heuser, K. Land, D. Hujo & M. Krüger (2022): *Educate complex C programming artefacts for robotics to mechanical engineers freshmen – Array, Pointer, Loop*. In: *2022 IEEE 18th International Conference on Automation Science and Engineering (CASE)*, pp. 2198–2204, doi:10.1109/CASE49997.2022.9926521.

[16] Alaeeddine Yassine, Driss Chenouni, Mohammed Berrada & Ahmed Tahiri (2017): *A Serious Game for Learning C Programming Language Concepts Using Solo Taxonomy*. *International Journal of Emerging Technologies in Learning (iJET)* 12(03), p. pp. 110–127, doi:10.3991/ijet.v12i03.6476. Available at `https://online-journals.org/index.php/i-jet/article/view/6476`.

Finite Automata for Efficient Graph Recognition

Frank Drewes
Umeå universitet
SE-90187 Umeå
Sweden
drewes@cs.umu.se

Berthold Hoffmann
Universität Bremen
D-28334 Bremen
Germany
hof@uni-bremen.de

Mark Minas
Universität der Bundeswehr München
D-85577 Neubiberg
Germany
mark.minas@unibw.de

Engelfriet and Vereijken have shown that linear graph grammars based on hyperedge replacement generate graph languages that can be considered as interpretations of regular string languages over typed symbols. In this paper we show that finite automata can be lifted from strings to graphs within the same framework. For the efficient recognition of graphs with these automata, we make them deterministic by a modified powerset construction, and state sufficient conditions under which deterministic finite graph automata recognize graphs without the need to use backtracking.

1 Introduction

Engelfriet and Vereijken [17] have shown that linear graph grammars based on hyperedge replacement can be considered as interpretations of linear string grammars: typed ("doubly ranked") symbols of an alphabet are interpreted as basic graphs that have front and rear interfaces of nodes, and string concatenation is interpreted as the composition of two graphs by gluing the rear of the first to the front of the second graph. Graph languages constructed in this way are of bounded pathwidth, and are thus potentially more efficiently recognizable than general hyperedge replacement languages, which are known to be NP-complete. However, without additional restrictions even these graph languages are NP-complete [1].

In this paper we study how finite automata over graph symbols can be interpreted to recognize graph languages efficiently. Given a graph as input, the transitions of such an automaton consume the graph step by step while changing the state of the automaton, in the end reaching either an accepting or a rejecting state. If the automaton used is nondeterministic, a naive decision procedure for determining whether the input graph is accepted would have to use backtracking. While we show in this paper that the transition relation can be made deterministic by a modified powerset construction, backtracking may still be needed. This is due to the fact that, whereas a string starts with a unique first symbol to be read by the first transition of the computation of a deterministic finite automaton, in general several alternative basic graphs represented by that symbol may be spelled off at the front of a graph. We provide two sufficient criteria under which automata can choose between them without the need to backtrack. These criteria resemble similar criteria known from efficient parsing algorithms for context-free hyperedge replacement languages [13, 14].

Work on efficient parsing algorithms for grammars started in the late 1980s, initiated by the realization that, in general, the graph languages generated by these grammars can be NP-complete [1, 28]. Early polynomial algorithms were based on restrictions which either ensure efficiency of the well-known Cocke-Younger-Kasami algorithm (adapted to hyperedge replacement grammars) or make sure that the derivation trees of generated graphs mirror a unique

J. Endrullis, D. Grzelak, T. Heindel and J. Kosiol (Eds):
GCM 2023 and GCM 2024
EPTCS 417, 2025, pp. 134–156, doi:10.4204/EPTCS.417.8

recursive decomposition of graphs into smaller and smaller subgraphs, see [29, 31, 10]. Later work on parsing hyperedge replacement languages include [8, 21] as well as the authors' own work on top-down and bottom-up parsers for these languages; see, e.g., [11, 12, 25, 14, 15]. For a more extensive overview of work on efficient parsing for graph grammars, including other types of grammars than those based on hyperedge replacement, see [14, Section 10].

For the efficient recognition of graph languages, finite automata have not attained the importance that they have for string languages. Some early work exists on finite automata for algebraic structures [2, 22], rooted directed acyclic graphs [32], and infinite directed acyclic graphs [27]. Brandenburg and Skodinis have devised finite automata for linear node replacement grammars [6]. Bozapalidis and Kalampakas have studied automata on the hypergraphs of Engelfriet and Vereijken [17] in an algebraic setting [5, 26]. And, Brugging, König *et al.* have recast these hypergraphs as cospans [3], and have shown that finite automata defining hypergraph languages are equivalent to Courcelle's recognizable hypergraph languages [9]; later these results have been generalized to hereditary pushout categories [7]. All this work has focussed on *defining* the graph languages accepted by finite automata (e.g., by composing the graphs labeling the edges passed on a walk through the transition diagrams of the automata), rather then on *recognizing* graph languages in the sense of efficiently deciding their membership problem. In particular, the efficiency of the recognition process has not been a matter of concern. In this paper, we focus on this aspect, which is of interest because the analysis of graphs by finite automata corresponds intuitively to a lexical analysis in the traditional string-based setting.

The technical contributions of this paper revolve mainly around different aspects of coping with nondeterminism, which differs in subtle ways from the string case. Given a nondeterministic automaton over graph symbols, we first apply a powerset construction to it (Section 4.1). Interestingly, this construction does not always result in a deterministic automaton. We show, however, that the resulting automaton is indeed deterministic if the construction starts from an unambiguous automaton, a condition that can always be fulfilled (Section 4.2). Next, as mentioned above, even a deterministic automaton is only deterministic at the string level. As the edges of a graph do not come in a predefined processing order, recognition still requires backtracking in general: anytime in the process, several different transitions may be applicable, and a given transition may have several alternative edges it may consume. We deal with these problems in Sections 4.3 and 4.4, resulting in linear-time recognition without backtracking.

The remainder of this paper is structured as follows. We define graphs with front and rear interfaces and their composition in Section 2. In Section 3, we introduce finite automata over graph symbols and define how they can be used to recognize graph languages. Section 4 contains the main technical contributions, as described above. In Section 5, we conclude the paper and indicate directions of future research.

2 Graphs and Graph Composition

We let $\mathbb{N}$ denote the set of non-negative integers, and $[n]$ the set $\{1,\dots,n\}$ for all $n \in \mathbb{N}$. A^* denotes the set of all finite sequences over a set A; the empty sequence is denoted by ε, $A^* \setminus \{\varepsilon\}$ by A^+, and the length of a sequence α by $|\alpha|$. For a sequence $s = a_1 \cdots a_n$ with $n \in \mathbb{N}$ and $a_1,\dots,a_n \in A$, we let $[s] = \{a_1,\dots,a_n\}$ and $s(i) = a_i$ for all $n \in \mathbb{N}$. For $f\colon A \to B^*$, we let $f(a,i) = f(a)(i)$ if $a \in A$ and $i \in [|f(a)|]$. We silently extend functions to sequences and let $f(a_1 \cdots a_n) = f(a_1) \cdots f(a_n)$, for all $n \in \mathbb{N}$, $a_1,\dots,a_n \in A$, and all functions $f\colon A \to B$. As usual,

given a binary relation $R \subseteq A \times B$, we denote the transitive closure by R^+ and the transitive reflexive closure by R^*. Further, $dom(R)$ denotes the domain of R, i.e., $dom(R) = \{a \in A \mid \exists b \in B\colon (a,b) \in R\}$. Finally, we let $R(a) = \{b \in B \mid (a,b) \in R\}$ for every $a \in A$.

We consider edge-labeled hypergraphs (which we simply call graphs). Like Habel in [23], we supply them with a front and a rear interface, each being a sequence of nodes.[1]

For the labeling of edges, we consider a ranked alphabet $(\Sigma, rank)$ consisting of a set of symbols and a function $rank\colon \Sigma \to \mathbb{N}$ which assigns a rank to each symbol $a \in \Sigma$. The pair $(\Sigma, rank)$ is usually identified with Σ, keeping $rank$ implicit.

Definition 2.1 (Graph)**.** A *graph* (over Σ) is a tuple $G = (\dot{G}, \bar{G}, att_G, lab_G, front_G, rear_G)$, where $\dot{G}$ and $\bar{G}$ are disjoint finite sets of *nodes* and *edges*, respectively, the function $att_G\colon \bar{G} \to \dot{G}^*$ attaches repetition-free sequences of nodes to edges, the function $lab_G\colon G \to \Sigma$ labels edges with symbols in such a way that $|att_G(e)| = rank(lab_G(e))$ for every edge $e \in \bar{G}$, and the repetition-free node sequences $front_G, rear_G \in \dot{G}^*$ specify the *front* and *rear* interface nodes.

The lengths of front and rear interfaces classify graphs as follows: A graph G *has type* (i,j) if $|front_G| = i$ and $|rear_G| = j$; we then write $type(G) = (i,j)$. $\mathbb{G}_\Sigma^{(i,j)}$ denotes the set of all graphs of type (i,j), and $\mathbb{G}_\Sigma = \bigcup_{i,j \in \mathbb{N}} \mathbb{G}_\Sigma^{(i,j)}$ denotes the set of all graphs over Σ, regardless of type.

For graphs G and H, a *morphism* $m\colon G \to H$ is a pair $m = (\dot{m}, \bar{m})$ of functions $\dot{m}\colon \dot{G} \to \dot{H}$ and $\bar{m}\colon \bar{G} \to \bar{H}$ that preserve attachments and labels, i.e., $att_H(\bar{m}(e)) = \dot{m}(att_G(e))$ and $lab_H(\bar{m}(e)) = lab_G(e)$ for all $e \in \bar{G}$. (Note that fronts and rears need not be preserved.) The morphism m is *injective* or *surjective* if both $\dot{m}$ and $\bar{m}$ have this property, and a *subgraph inclusion* of G in H if $m(x) = x$ for every node and edge x in G; then we write $G \subseteq H$. G and H are *isomorphic*, $G \cong H$, if there exists a surjective and injective morphism $m\colon G \to H$ such that $front_H = m(front_G)$ and $rear_H = m(rear_G)$.

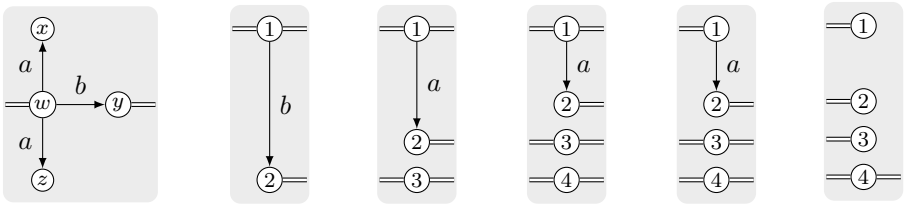
Consider $a \in \Sigma$, $n \in \mathbb{N}$, and repetition-free sequences $\varphi, \varrho \in [n]^*$ such that $[\varphi] \cup [rank(a)] = [n]$ (where $[\varphi]$ and $[rank(a)]$ are not necessarily disjoint). Then $\langle a \rangle_\varrho^\varphi$ denotes the graph A with $\dot{A} = [n]$, $\bar{A} = \{e\}$, $lab_A(e) = a$, $att_A(e) = 1 \cdots rank(a)$, $front_A = \varphi$, and $rear_A = \varrho$, whereas $\langle \varepsilon \rangle_\varrho^{(n)}$ denotes the discrete graph B with $\bar{B} = \varnothing$, $\dot{B} = [n]$, $front_B = 1 \cdots n$, and $rear_A = \varrho$. We call $\langle a \rangle_\varrho^\varphi$ an *atom* and $\langle \varepsilon \rangle_\varrho^{(n)}$ a *blank*.

Note that $\langle \varepsilon \rangle_\varepsilon^{(0)}$ is the empty graph. Further note that no atom has any node that neither is a front node, nor is it attached to its only edge. In particular, every rear node is also a front node or attached to the unique edge of the atom, or both. Moreover, all nodes of a blank occur in its front interface. Finally recall that our objective is efficient recognition of graphs, which tries to compose an input graph from a sequence of atoms and a blank. The requirements on front and rear interfaces differ because recognition will process graphs from front to rear.

Example 2.1. Figure 1 shows a "star" graph, atoms $\langle b \rangle_{12}^1$, $\langle a \rangle_{123}^{13}$, $\langle a \rangle_{1234}^{134}$, and $\langle a \rangle_{234}^{134}$ using the symbols a and b, both of rank 2, and the blank $\langle \varepsilon \rangle_4^{(4)}$. As usual, we represent nodes by circles and binary edges by arrows. Front interface nodes are connected with double lines to the left border of the graph, rear interface nodes with double lines to its right border. Front and rear nodes are ordered from top to bottom; the "star" graph, e.g., has the front interface w and the rear interface y.

We follow Engelfriet and Vereijken [17] in defining the composition of graphs in a way resembling string concatenation.

[1] Other than in [23], however, we do not divide the attached nodes of edges into sources and targets.



"Star" graph $\langle b\rangle_{12}^{1}$ $\langle a\rangle_{123}^{13}$ $\langle a\rangle_{1234}^{134}$ $\langle a\rangle_{234}^{134}$ $\langle \varepsilon\rangle_{4}^{(4)}$

Figure 1: A "star" graph, four atoms, and a blank.

Definition 2.2 (Graph Composition)**.** Let $G \in \mathbb{G}_\Sigma^{(i,k)}$ and $H \in \mathbb{G}_\Sigma^{(k,j)}$. We assume for simplicity that $\mathit{rear}_G = \mathit{front}_H$, $\dot{G} \cap \dot{H} = [\mathit{rear}_G] = [\mathit{front}_H]$, and $\bar{G} \cap \bar{H} = \varnothing$ (otherwise an appropriate isomorphic copy of G or H is used). The (*typed*) *composition* $G \odot H$ of G and H is the graph C such that $\dot{C} = \dot{G} \cup \dot{H}$, $\bar{C} = \bar{G} \cup \bar{H}$, $\mathit{att}_C = \mathit{att}_G \cup \mathit{att}_H$, $\mathit{lab}_C = \mathit{lab}_G \cup \mathit{lab}_H$, $\mathit{front}_C = \mathit{front}_G$, and $\mathit{rear}_C = \mathit{rear}_H$. Thus $C \in \mathbb{G}_\Sigma^{(i,j)}$.

Note that the composition $G \odot H$ is defined on concrete graphs if the assumptions in Definition 2.2 are satisfied, but is only defined up to isomorphism if an isomorphic copy of G or H needs to be taken. To avoid unnecessary technicalities, we shall assume that these assumptions are indeed satisfied whenever convenient.

Graphs can be constructed from a finite set of basic graphs using composition and disjoint union [18, 4] (where the latter concatenates the fronts and rears of the two graphs involved). If we just use composition, we need finitely many basic graphs per pair of front and rear interfaces. Instead of the simpler "atomic graphs" proposed in [17] and by Blume et al. in [3], we use atoms and blanks here.

Lemma 2.3. *Every graph G whose isolated nodes all occur in its front interface is of the form $A_1 \odot \cdots \odot A_n \odot B$ where $n = |\bar{G}|$, $A_1, \ldots, A_n$ are atoms, and B is a blank.*

Proof. Let $G \in \mathbb{G}_\Sigma$ with $|\bar{G}| = n$ such that every isolated node of G occurs in $[\mathit{front}_G]$. We prove by induction on n that there are atoms $A_1, \ldots, A_n$ and a blank B such that $G \cong A_1 \odot \cdots \odot A_n \odot B$. If $n = 0$, all nodes of G are isolated, and hence $\dot{G} = [\mathit{front}_G]$. Then $G \cong \langle \varepsilon \rangle_\varrho^{(m)}$ where $m = |\dot{G}|$ and ϱ is a sequence of length $k = |\mathit{rear}_G|$ such that $\mathit{rear}_G(i) = \mathit{front}_G(\varrho(i))$ for all $i \in [k]$.

Now consider $n > 0$ and assume, as an induction hypothesis, that the proposition holds for all graphs with $n-1$ edges. Pick any edge of G, say $e \in \bar{G}$, and let G' be the graph obtained from G by removing e and setting $\mathit{front}_{G'}$ to any permutation of $[\mathit{front}_G] \cup [\mathit{att}_G(e)]$. Since G' has $n-1$ edges, there are $n-1$ atoms $A_2, \ldots, A_n$ and a blank B such that $G' \cong A_2 \odot \cdots \odot A_n \odot B$ by the induction hypothesis. Now let $m = |\mathit{front}_G|$, $k = |\mathit{front}_{G'}|$, and $A_1 = \langle \mathit{lab}_G(e) \rangle_\varrho^\varphi$ where ϱ is a permutation of $[k]$, and φ is a sequence of length m such that the following holds: $\mathit{front}_{G'}(i) = \mathit{att}_G(e, \varrho(i))$ for all $i \in [k]$ with $\mathit{front}_{G'}(i) \in [\mathit{att}_G(e)]$. Moreover, $\varphi(i) = \varrho(j)$ if $\mathit{front}_G(i) = \mathit{front}_{G'}(j)$, for all $i \in [m]$ and $j \in [k]$. It is easy to see that $G \cong A_1 \odot G' \cong A_1 \odot \cdots \odot A_n \odot B$. □

Note that the size of interfaces required to build G according to Lemma 2.3 depends on G. Hence, the lemma does not contradict the fact, mentioned in Section 1, that the set of all graphs composed from a given finite set of atoms and blanks is of bounded pathwidth.

Example 2.2. Following the construction in the proof, the "star" graph G in Figure 1 can be composed by $G \cong \langle b\rangle_{12}^{1} \odot \langle a\rangle_{123}^{13} \odot \langle a\rangle_{1234}^{134} \odot \langle \varepsilon\rangle_{4}^{(4)}$, all shown in Figure 1. However, this is not the only composition. A much simpler one is $G \cong \langle a\rangle_{1}^{1} \odot \langle a\rangle_{1}^{1} \odot \langle b\rangle_{2}^{1}$.

The example shows that, unlike a string, a graph can be composed from atoms and blanks in different ways. Moreover, the size of atoms and blanks can also vary substantially.

3 Finite Automata over Graph Symbols

We now follow the idea of Engelfriet and Vereijken [17] to make use of the close resemblance of graphs under (typed) composition and strings under concatenation, and denote graphs by strings. Each symbol is interpreted as a graph, and thus a string translates into a graph, provided that the types of composed graphs fit. To this end, we type each symbol by a pair (i,j) – the type of the graph it will represent.

A *typed alphabet* is a pair $(\mathbf{\Theta}, \mathit{type})$ consisting of a (possibly infinite) set $\mathbf{\Theta}$ of symbols and a function $\mathit{type}\colon \mathbf{\Theta} \to \mathbb{N}\times\mathbb{N}$ which assigns a pair $\mathit{type}(a)$ of front and rear ranks to each symbol $a \in \mathbf{\Theta}$. The pair $(\mathbf{\Theta}, \mathit{type})$ is usually identified with $\mathbf{\Theta}$, and type is kept implicit.

A string $w = a_1 \cdots a_n \in \mathbf{\Theta}^+$ is *typed* if there are $k_0, \ldots, k_n \in \mathbb{N}$ such that $\mathit{type}(a_i) = (k_{i-1}, k_i)$ for all $i \in [n]$. We let $\mathit{type}(w) = (k_0, k_n)$. The set of all typed strings over $\mathbf{\Theta}$ is written $\mathbf{\Theta}^{\oplus}$. The concatenation $u \cdot v$ of typed strings $u, v \in \mathbf{\Theta}^{\oplus}$ with $\mathit{type}(u) = (i,j)$ and $\mathit{type}(v) = (m,n)$ is only defined if $j = m$.

Note that ordinary (untyped) alphabets and strings over them can be considered as special cases of typed ones by setting $\mathit{type}(a) = (1,1)$ for every symbol a. Further note that there is no empty typed string $\varepsilon \in \mathbf{\Theta}^+$ because its type would be undefined. Instead, we introduce so-called *blanks* below.

For a typed alphabet $\mathbf{\Theta}$ and a ranked alphabet Σ, an *interpretation operator* $[\![\cdot]\!] : \mathbf{\Theta} \to \mathbb{G}_\Sigma$ assigns a graph $[\![a]\!] \in \mathbb{G}_\Sigma$ with $\mathit{type}([\![a]\!]) = \mathit{type}(a)$ to each symbol $a \in \mathbf{\Theta}$. We extend $[\![\cdot]\!]$ to typed strings over $\mathbf{\Theta}$ by $[\![a_1 \cdots a_n]\!] \cong [\![a_1]\!] \odot \cdots \odot [\![a_n]\!]$ where $a_i \in \mathbf{\Theta}$ for $i \in [n]$.

Every interpretation operator $[\![\cdot]\!] : \mathbf{\Theta} \to \mathbb{G}_\Sigma$ defines a congruence relation $\sim$ on typed strings, as follows: for all $u, v \in \mathbf{\Theta}^{\oplus}$, $u \sim v$ if and only if $[\![u]\!] \cong [\![v]\!]$.

Given a ranked alphabet Σ, the *canonical alphabet* $(\mathbf{\Theta}_\Sigma, \mathit{type})$ is the typed alphabet given by

$$\begin{aligned}
\mathbf{\Theta}_\Sigma &= \{a_\varrho^\varphi \mid \langle a\rangle_\varrho^\varphi \text{ is an atom in } \mathbb{G}_\Sigma\} \cup \mathcal{B} \\
\mathcal{B} &= \{\varepsilon_\varrho^{(n)} \mid \langle \varepsilon\rangle_\varrho^{(n)} \text{ is a blank in } \mathbb{G}_\Sigma\} \\
\mathit{type}(a_\varrho^\varphi) &= (|\varphi|, |\varrho|) \quad \text{for all } a_\varrho^\varphi \in \mathbf{\Theta}_\Sigma \\
\mathit{type}(\varepsilon_\varrho^{(n)}) &= (n, |\varrho|) \quad \text{for all } \varepsilon_\varrho^{(n)} \in \mathcal{B}.
\end{aligned}$$

The *canonical interpretation* of $\mathbf{\Theta}_\Sigma$ is given by $[\![a_\varrho^\varphi]\!] = \langle a\rangle_\varrho^\varphi$ for all $a_\varrho^\varphi \in \mathbf{\Theta}_\Sigma \setminus \mathcal{B}$ and $\left[\!\!\left[\varepsilon_\varrho^{(n)}\right]\!\!\right] = \langle \varepsilon\rangle_\varrho^{(n)}$ for all $\varepsilon_\varrho^{(n)} \in \mathcal{B}$. We call all symbols in $\mathcal{B}$ *blank symbols* (or just *blanks*), and abbreviate $\varepsilon_\varrho^{(n)}$ with $\varrho = 1 \cdots n$ as $\varepsilon^{(n)}$. Note that $\mathcal{B} \subseteq \mathbf{\Theta}_\Sigma$ is infinite and independent of Σ.

We note the following immediate consequences of the definitions above:

Fact 3.1. *The following holds for every ranked alphabet Σ and its canonical alphabet $\mathbf{\Theta}_\Sigma$:*

1. *Let $a, b \in \mathbf{\Theta}_\Sigma$ with $\mathit{type}(a) = (m,n)$ and $\mathit{type}(b) = (n,k)$ (for some $m,n,k \in \mathbb{N}$) be such that $\{a,b\} \cap \mathcal{B} \neq \varnothing$. Then $\mathbf{\Theta}_\Sigma$ contains a symbol c such that $ab \sim c$. In particular, $\mathit{type}(c) = (m,k)$. (Note that $c \in \mathcal{B}$ if both a and b are blanks, and $c \in \mathbf{\Theta}_\Sigma \setminus \mathcal{B}$ otherwise.)*

2. For all $m,n \in \mathbb{N}$ and $a \in \mathbf{\Theta}_\Sigma$ with $type(a) = (n,m)$, we have $\varepsilon^{(n)}a \sim a \sim a\varepsilon^{(m)}$.

By requiring that finite automata respect types, we obtain finite automata over a finite typed alphabet Θ.

Definition 3.2 (Finite Automaton)**.** Let Θ be a finite typed alphabet. A *finite automaton* over Θ is a tuple $\mathfrak{A} = (\Theta, Q, \Delta, q_0, F)$ such that Q is a ranked alphabet of *states*, $\Delta \subseteq (Q \times \Theta \times Q)$ is a set of *transitions* (q,a,q') such that $type(a) = (i,j)$ implies that $rank(q) = i$ and $rank(q') = j$, $q_0 \in Q$ is the *initial state*, and $F \subseteq Q \setminus \{q_0\}$ is a non-empty set of *final states* such that $rank(q) = rank(q')$ for all $q, q' \in F$.

We let $type(\mathfrak{A}) = (m,n)$ where $m = rank(q_0)$ and $n = rank(q)$ for all $q \in F$. $\mathfrak{A}$ is *deterministic* if, for each pair of transitions $(p,a,q),(p',b,q') \in \Delta$, $p = p'$ and $a = b$ implies $q = q'$.

A *configuration* of $\mathfrak{A}$ is a pair $(q,w) \in Q \times \Theta^*$ consisting of the *current state* q and the *remaining input* w. It is *initial* if $q = q_0$, and *accepting* if $q \in F$ and $w = \varepsilon$. Note that configurations are defined for arbitrary $w \in \Theta^*$, i.e., w is not necessarily typed.

A transition $\delta = (q,a,q') \in \Delta$, where a is a symbol in Θ, defines *moves* $(q,aw) \vdash_\delta (q',w)$ for all strings $w \in \Theta^*$. We write $(q,w) \vdash_{\mathfrak{A}} (q',w')$ if $(q,w) \vdash_\delta (q',w')$ for some transition $\delta \in \Delta$.

The *language* accepted by $\mathfrak{A}$ is defined as usual:

$$\mathcal{L}(\mathfrak{A}) = \{w \in \Theta^* \mid \exists q \in F : (q_0,w) \vdash_{\mathfrak{A}}^* (q,\varepsilon)\}.$$

Note that $\mathcal{L}(\mathfrak{A}) \subseteq \Theta^\oplus$ because transitions are typed and $q_0 \notin F$. In fact, we have $type(w) = type(\mathfrak{A})$ for all $w \in \mathcal{L}(\mathfrak{A})$.

Associating a graph interpretation operator with Θ, we can recognize graph languages in the obvious way: a graph G is accepted by an automaton $\mathfrak{A}$ if $G \cong [\![w]\!]$ for some string $w \in \mathcal{L}(\mathfrak{A})$. To enable our automata to work directly on graphs, we use *graph configurations* and the corresponding moves:

Definition 3.3 (Graph configurations and moves)**.** Let $\mathfrak{A} = (\Theta, Q, \Delta, q_0, F)$ be a finite automaton with $\Theta \subseteq \mathbf{\Theta}_\Sigma$. A pair (q,G) consisting of a state $q \in Q$ and a graph $G \in \mathbb{G}_\Sigma$ is a *graph configuration*, or simply *configuration* if the context prevents confusion.

A transition $\delta = (q,a,q') \in \Delta$ can be applied to a graph $G \in \mathbb{G}_\Sigma$ if there are graphs $G', G_a \in \mathbb{G}_\Sigma$ such that $G = G_a \odot G'$ and $G_a \cong [\![a]\!]$. We then call $(q,G) \Vdash_\delta (q',G')$ a *move* (using δ). We write $(q,G) \Vdash_{\mathfrak{A}} (q',G')$ if $(q,G) \Vdash_\delta (q',G')$ for some transition $\delta \in \Delta$. The set of *acceptable configurations* of $\mathfrak{A}$ with $type(\mathfrak{A}) = (m,n)$ is

$$\mathcal{C}_{\mathfrak{A}} = \{(q,G) \in Q \times \mathbb{G}_\Sigma \mid \exists q' \in F, G' \in \mathbb{G}_\Sigma \colon (q,G) \Vdash_{\mathfrak{A}}^* (q',G') \text{ and } G' \cong \langle \varepsilon \rangle_{1\ldots n}^{(n)}\}.$$

The graph language accepted by $\mathfrak{A}$ is then

$$\mathcal{L}_G(\mathfrak{A}) = \{G \in \mathbb{G}_\Sigma \mid (q_0,G) \in \mathcal{C}_{\mathfrak{A}}\}.$$

Obviously (and provable by a straightforward induction), we have $\mathcal{L}_G(\mathfrak{A}) = \{[\![w]\!] \mid w \in \mathcal{L}(\mathfrak{A})\}$. However, given a graph $G \in \mathcal{L}_G(\mathfrak{A})$, it is usually not the case that $w \in \mathcal{L}(\mathfrak{A})$ for all $w \in \Theta^\oplus$ such that $[\![w]\!] = G$, as demonstrated in the following example. This turns efficient graph recognition with finite automata into a nontrivial problem.

Example 3.1. Let us consider the graph language of "stars" where each "star" consists of a center node, which is the only front interface node, and at least two satellite nodes, which are connected by binary edges with the center node. Just one of the edges is labeled with b, the

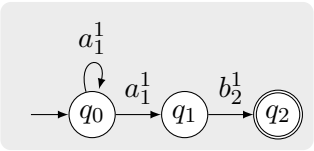


Figure 2: The finite automaton $\mathfrak{S}$ recognizing the graph language of "stars".

others with a, and the rear interface consists of just the satellite node attached to the b-labeled edge. Figure 1 shows such a "star", and Figure 2 the finite automaton $\mathfrak{S}$ recognizing this graph language. As usual, we draw finite automata with circles as states and arrows as transitions. The initial state is indicated by an incoming arrow from nowhere, final states by double borders. $\mathfrak{S}$ is nondeterministic since it contains a transition (q_0, a_1^1, q_0) as well as (q_0, a_1^1, q_1).

One can see that the "star" G in Figure 1 is a member of $\mathcal{L}_G(\mathfrak{S})$ because the string $w = a_1^1\, a_1^1\, b_2^1$ is recognized by $\mathfrak{S}$ and $G \cong [\![w]\!]$. Note that $[\![w]\!] \cong [\![a_1^1]\!] \odot [\![a_1^1]\!] \odot [\![b_2^1]\!] = \langle a \rangle_1^1 \odot \langle a \rangle_1^1 \odot \langle b \rangle_2^1$, which is one of the compositions of G shown in Example 2.2, whereas, e.g., the string $b_{12}^1\, a_{123}^{13}\, a_{1234}^{134}\, \varepsilon_4^{(4)}$ cannot be recognized by $\mathfrak{S}$.

4 Efficient Graph Recognition with Finite Automata

Let us now use a finite automaton for the recognition of graphs. To achieve efficiency, we must avoid nondeterminism. To see this, consider a nondeterministic automaton and a situation where we have reached a configuration with a state that has several outgoing transitions reading the same symbol. This means that all of them are applicable whenever one of them can be applied. Consequently, we must try one of them first, and if this choice leads into a dead end later, we are forced to backtrack and then try the next one. Backtracking usually leads to exponential running times, and should thus be avoided for efficient recognition.

Note that blank transitions, that is, transitions that read a blank, can lead to nondeterminism, even if the automaton is in fact deterministic. To understand this, consider a state with at least two outgoing transitions, one of them a blank transition. Similarly to an epsilon transition in an ordinary finite automaton, a blank transition (of the right type) can always be applied. So whenever one of the other transitions is applicable, one must choose between that transition and the blank transition. If one picks the "wrong" one that leads into a dead end, backtracking is necessary in order to choose the other one.

For efficient graph recognition with finite automata, we first develop an extension of the well-known powerset construction to turn a finite automaton over a subset of the canonical alphabet into a deterministic one without blank transitions in places where they may trigger backtracking.

4.1 Powerset Construction for Finite Automata

General Assumption. Throughout this subsection, we consider a fixed (ranked) alphabet Σ, its canonical alphabet $\boldsymbol{\Theta}_\Sigma$, and a finite automaton $\mathfrak{A} = (\Theta, Q, \Delta, q_0, F)$ over a typed alphabet $\Theta \subset \boldsymbol{\Theta}_\Sigma$. We also let $(m, n) = \mathit{type}(\mathfrak{A})$.

Algorithm 1 extends the well-known powerset construction and computes a new automaton whose states are built from sets of states of the input automaton. As mentioned above, blank

Algorithm 1: Powerset construction for a finite automaton.

Input : Finite automaton $\mathfrak{A} = (\Theta, Q, \Delta, q_0, F)$ with $\Theta \subset \mathbf{\Theta}_\Sigma$.
Output: Powerset automaton $\mathfrak{A}' = (\Theta', Q', \Delta', S_0, F')$.

1 $S_0 \leftarrow Cl(q_0)$ and $S_f \leftarrow \varnothing$
2 $Q' \leftarrow \{S_0, S_f\}$ and $F' \leftarrow \{S_f\}$ where $(rank(S_0), rank(S_f)) = type(\mathfrak{A})$
3 $\Delta' \leftarrow \varnothing$
4 *frontier* $\leftarrow \{S_0\}$
5 **while** *frontier* $\neq \varnothing$ **do**
6 select and remove any X from *frontier*
7 **foreach** $(\beta, q) \in X$ *such that* $q \in F$ **do** add (X, β, S_f) to Δ'
8 $\psi \leftarrow \{(a, q') \in (\mathbf{\Theta} \setminus \mathcal{B}) \times Q \mid (\beta, q) \in X,\ (q, b, q') \in \Delta,$ and $a \sim \beta b\}$
9 **foreach** $a_0 \in dom(\psi)$ **do**
10 let $type(a_0) = (i, j)$
11 $Y \leftarrow Cl(\psi(a_0))$
12 **if** $Y = \beta' Y'$ *for some* $Y' \in Q'$ *and* $\beta' \in \mathcal{B}$ *with* $type(\beta') = (j, j)$ **then**
13 add (X, a_1, Y') to Δ' for some $a_1 \in \Theta \setminus \mathcal{B}$ such that $a_1 \sim a_0 \beta'$
14 **else**
15 let $rank(Y) = j$ and add Y to Q' as well as to *frontier*
16 add (X, a_0, Y) to Δ'
17 $\Theta' \leftarrow \{a \in \mathbf{\Theta}_\Sigma \mid \exists S, S'' \in Q' : (S, a, S') \in \Delta'\}$

transitions resemble epsilon transitions that "read" the empty word. However, blanks are more complicated because they must take nodes into account; remember that the rear interface of a blank can contain a subset of its nodes in any order (cf. Figure 1). Therefore, Algorithm 1 considers pairs of the form (β, q) where β is a blank and q a state of the input automaton. Informally speaking, such a pair indicates that one can reach state q after reading β. Note that β can be $\varepsilon^{(i)}$ for an appropriate i, that is, an identity without any effect (which is comparable to ε in the string case). The states of the computed automaton then consist of sets of such pairs.

When building these sets, the algorithm must follow any sequence of blank transitions and combine their blanks into a single blank, which is always possible thanks to Fact 3.1. We call the set of all states (together with these blanks) that can be reached from a state $q \in Q$ by a sequence of blank transitions the *closure* of q. It is defined as

$$Cl(q) = \{(\beta, q') \in \mathcal{B} \times Q \mid \exists \sigma \in \mathcal{B}^* \colon (q, \sigma) \vdash_{\mathfrak{A}}^* (q', \varepsilon) \text{ and } \varepsilon^{(i)} \beta \sim \varepsilon^{(i)} \sigma\}$$

where $rank(q) = i$. The condition $\varepsilon^{(i)} \beta \sim \varepsilon^{(i)} \sigma$ in this definition can be simplified to $\beta \sim \sigma$ unless $\sigma = \varepsilon$. If $\sigma = \varepsilon$ then $\varepsilon^{(i)}$ is required because $[\![\varepsilon]\!]$ is undefined. Note also that $(\varepsilon^{(i)}, q) \in Cl(q)$, and that $(\beta, q') \in Cl(q)$ implies $type(\beta) = (i, rank(q'))$.

We extend the definition of closures to sets of states by defining $Cl(M) = \bigcup_{q \in M} Cl(q)$ for all $M \subseteq Q$ as used in Line 11. For every set $C \subseteq \mathcal{B} \times Q$ and $i \in \mathbb{N}$ such that $(\beta', q) \in C$ implies $type(\beta') = (i, rank(q))$, and every blank $\beta \in \mathcal{B}$ of type (i, i), we define

$$\beta C = \{(\beta'', q) \in \mathcal{B} \times Q \mid \exists \beta' \in \mathcal{B} \colon (\beta', q) \in C \text{ and } \beta'' \sim \beta \beta'\}$$

as used in Line 12. Note that a blank β'' with $\beta'' \sim \beta\beta'$ always exists, thanks to Fact 3.1.1.

We now prove the correctness of Algorithm 1 by first showing that it always terminates and produces a proper finite automaton as a result. After that, we will show that the produced powerset automaton is equivalent to the input automaton.

Lemma 4.1. *Algorithm 1 terminates for every finite automaton* $\mathfrak{A} = (\Theta, Q, \Delta, q_0, F)$ *and outputs a finite automaton.*

Proof. We use the notations established in Algorithm 1. In particular, $\mathfrak{A}' = (\Theta', Q', \Delta', S_0, F')$ is the output of the algorithm.

Every state in Q' is a subset of $\mathcal{B} \times Q$. The fact that Q and Θ are finite thus implies that Q' is finite. Since no element of Q' is added to *frontier* more than once, none of the loops of Algorithm 1 can run endlessly.

It remains to be shown that the transitions of $\mathfrak{A}'$ are correctly typed. For this, we consider any transition $\delta = (S, a, S') \in \Delta'$ and show that $\mathit{type}(a) = (\mathit{rank}(S), \mathit{rank}(S'))$. Transition δ must have been added in Lines 7, 13, or 16. Before we consider each of these cases, first note that each state $S \neq S_f$ is a non-empty set of pairs $(\beta, q) \in \mathcal{B} \times Q$ that satisfy $\mathit{type}(\beta) = (\mathit{rank}(S), \mathit{rank}(q))$ by the definition of closures, how ψ is computed in Line 8, and how $\mathit{rank}(S)$ is set in Lines 2 and 15. Further note that we have $\mathit{rank}(X) = i$ when Line 12 is reached, also by the definition of closures and how ψ is computed. We now distinguish the three cases where δ may have been added to Δ':

- Line 7: Since $(\beta, q) \in X$ and $q \in F$, $\mathit{type}(\beta) = (\mathit{rank}(X), \mathit{rank}(q)) = (\mathit{rank}(X), \mathit{rank}(S_f))$
- Line 13: $\mathit{rank}(Y') = j$ and $\mathit{type}(a_1) = (i, j)$ follow from $Y = \beta' Y'$, $\mathit{type}(\beta') = (j, j)$, and $\mathit{type}(a_0) = (i, j)$, and hence $\mathit{type}(a_1) = (\mathit{rank}(X), \mathit{rank}(Y'))$.
- Line 16: $\mathit{rank}(Y)$ is set to j, and hence $\mathit{type}(a_0) = (\mathit{rank}(X), \mathit{rank}(Y))$,

Hence, the transitions in Δ' are correctly typed, i.e., $\mathfrak{A}'$ is a finite automaton. □

To prove that the powerset automation $\mathfrak{A}'$ produced by Algorithm 1 is equivalent to its input automaton $\mathfrak{A}$, we now show that every accepting sequence of moves in $\mathfrak{A}$ has a corresponding sequence of moves in $\mathfrak{A}'$ (Lemma 4.3) and vice versa (Lemma 4.4). For proving Lemma 4.3, we will need the following auxiliary result. Informally speaking (and ignoring the fact that we are actually dealing with pairs (β, q) instead of just states q), Lemma 4.2 proves that every state reachable from a closure state by a blank transition is also contained in the closure:

Lemma 4.2. *Let* $q, q', q'' \in Q$, $\beta, \beta', \beta'' \in \mathcal{B}$, *and* $\delta = (q', \beta', q'') \in \Delta$. *If* $(\beta, q') \in \mathit{Cl}(q)$ *and* $\beta'' \sim \beta\beta'$ *then* $(\beta'', q'') \in \mathit{Cl}(q)$.

Proof. Let $\mathit{rank}(q) = i$. By the definition of $\mathit{Cl}(q)$, $(\beta, q') \in \mathit{Cl}(q)$ implies $(q, \sigma) \vdash_{\mathfrak{A}}^{*} (q', \varepsilon)$ and $\varepsilon^{(i)}\beta \sim \varepsilon^{(i)}\sigma$ for some $\sigma \in \mathcal{B}^*$. Consequently, $(q, \sigma\beta') \vdash_{\mathfrak{A}}^{*} (q', \beta') \vdash_{\delta} (q'', \varepsilon)$ and $\varepsilon^{(i)}\sigma\beta' \sim \varepsilon^{(i)}\beta\beta' \sim \varepsilon^{(i)}\beta''$, and thus $(\beta'', q'') \in \mathit{Cl}(q)$. □

Lemma 4.3. *Consider* $q \in Q$ *and* $w \in \Theta^*$ *such that* $(q_0, w) \vdash_{\mathfrak{A}}^{*} (q, \varepsilon)$. *Then there exist* $w' \in \Theta^*$, $S \in Q'$, *and* $\beta \in \mathcal{B}$ *such that* $(S_0, w') \vdash_{\mathfrak{A}'}^{*} (S, \varepsilon)$, $(\beta, q) \in S$, *and* $\varepsilon^{(m)} w \sim \varepsilon^{(m)} w' \beta$.[2]

Proof. We prove the proposition by induction over the length ℓ of the move sequence $(q_0, w) \vdash_{\mathfrak{A}}^{*} (q, \varepsilon)$. The proposition is clearly true for $\ell = 0$ (with $w = w' = \varepsilon$, $S = S_0$, and $\beta = \varepsilon^{(m)}$). For $\ell > 0$, we consider the sequence $(q_0, wa) \vdash_{\mathfrak{A}}^{\ell-1} (q, a) \vdash_{\delta} (q', \varepsilon)$ using transition $\delta = (q, a, q') \in \Delta$.

[2] Recall that $\mathit{type}(\mathfrak{A}) = (m, n)$.

By the induction hypothesis, there is a sequence $(S_0, w') \vdash^*_{\mathfrak{A}'} (S, \varepsilon)$ with $(\beta, q) \in S$ and $\varepsilon^{(m)} w \sim \varepsilon^{(m)} w'\beta$. We have to show that there is also a sequence $(S_0, w'a') \vdash^*_{\mathfrak{A}'} (S', \varepsilon)$ with $(\gamma, q') \in S'$ and $\varepsilon^{(m)} wa \sim \varepsilon^{(m)} w'a'\gamma$. We distinguish two cases:

- $a \in \mathcal{B}$: By Lines 2, 11, and 15 of Algorithm 1, $(\beta, q) \in Cl(q'') \subseteq S$ for some $q'' \in Q$. Let $\beta' \in \mathcal{B}$ such that $\beta' \sim \beta a$. We have $(\beta', q') \in Cl(q'') \subseteq S$ by Lemma 4.2 and $\varepsilon^{(m)} wa \sim \varepsilon^{(m)} w'\beta a \sim \varepsilon^{(m)} w'\beta'$ by the induction hypothesis and the definition of β'. This proves the proposition by choosing $a' = \varepsilon$ and $\gamma = \beta'$.
- $a \in \Theta \setminus \mathcal{B}$: Whenever an element is added to Q', it is also added to *frontier* (Line 4 and Line 15). Since $S \in Q'$ and *frontier* $= \varnothing$ when Algorithm 1 terminates, S must have been selected as X in Line 6 at some point. As $(\beta, q) \in S$ and $\delta = (q, a, q') \in \Delta$, there is a $b \in \Theta \setminus \mathcal{B}$ such that $b \sim \beta a$, $(b, q') \in \psi$ after executing Line 8, and $(\varepsilon^{(j)}, q') \in Y$ after executing Line 11, and after selecting b as a_0 in Line 9. Y is handled in one of two sub-cases selected in Lines 12 and 14:
  - Line 12, i.e., Q' already contains a similar nonterminal Y' such that $Y = \beta' Y'$ for some blank $\beta' \in \mathcal{B}$ with $type(\beta') = (j, j)$, and $\delta' = (X, b', Y') = (S, b', Y')$ is added to Δ' where $b' \in \Theta \setminus \mathcal{B}$ and $b' \sim b\beta' = a_0\beta'$. Consequently, $(S_0, w'b') \vdash^*_{\mathfrak{A}'} (S, b') \vdash_{\delta'} (Y', \varepsilon)$. Since $(\varepsilon^{(j)}, q') \in Y$, there exists $\beta'' \in \mathcal{B}$ such that $(\beta'', q') \in Y'$ and $\varepsilon^{(j)} \sim \beta'\beta''$. Therefore,

    $$\varepsilon^{(m)} wa \sim \varepsilon^{(m)} w'\beta a \sim \varepsilon^{(m)} w'b \sim \varepsilon^{(m)} w'b\varepsilon^{(j)} \sim \varepsilon^{(m)} w'b\beta'\beta'' \sim \varepsilon^{(m)} w'b'\beta''$$

    by the induction hypothesis, the construction of b and b', and the fact that $\varepsilon^{(j)} \sim \beta'\beta''$. This proves the proposition by choosing $a' = b'$, $\gamma = \beta''$, and $S' = Y'$.
  - Line 14, i.e., Y is not already in Q', and neither is there a similar one. Thus, Y is added to Q' and $\delta' = (X, b, Y) = (S, b, Y)$ is added to Δ'. Consequently, $(S_0, w'b) \vdash^*_{\mathfrak{A}} (S, b) \vdash_{\delta'} (Y, \varepsilon)$ and $\varepsilon^{(m)} wa \sim \varepsilon^{(m)} w'\beta a \sim \varepsilon^{(m)} w'b$ by the induction hypothesis and the construction of b. This proves the proposition with $a' = b$, $\gamma = \varepsilon$, and $S' = Y$. □

Lemma 4.4. *Let $S \in Q'$, $\beta \in \mathcal{B}$, $q \in Q$, and $w' \in \Theta^*$ be such that $(\beta, q) \in S$ and $(S_0, w') \vdash^*_{\mathfrak{A}'} (S, \varepsilon)$. Then there exists $w \in \Theta^*$ such that $(q_0, w) \vdash^*_{\mathfrak{A}} (q, \varepsilon)$ and $\varepsilon^{(m)} w \sim \varepsilon^{(m)} w'\beta$.*

Proof. We prove the proposition by induction over the length ℓ of the move sequence $(S_0, w') \vdash^*_{\mathfrak{A}'} (S, \varepsilon)$. For $\ell = 0$, we have $(S_0, \varepsilon) \vdash^0_{\mathfrak{A}'} (S_0, \varepsilon)$. Consider any $(\beta, q) \in S_0$. $S_0 = Cl(q_0)$ implies $(q_0, \sigma) \vdash^*_{\mathfrak{A}} (q, \varepsilon)$ for some $\sigma \in \mathcal{B}^*$ such that $\varepsilon^{(m)} \sigma \sim \varepsilon^{(m)} \beta$. This proves the proposition by choosing $w' = \varepsilon$ and $w = \sigma$.

For $\ell > 0$, the sequence $(S_0, w') \vdash^*_{\mathfrak{A}'} (S, \varepsilon)$ has the form $(S_0, w'_0 a) \vdash^{\ell-1}_{\mathfrak{A}'} (S', a) \vdash_\delta (S, \varepsilon)$, where $w' = w'_0 a$ and $\delta = (S', a, S) \in \Delta'$. By the induction hypothesis, for every $(\gamma, q') \in S'$, there is $w_0 \in \Theta^*$ such that $(q_0, w_0) \vdash^*_{\mathfrak{A}} (q', \varepsilon)$ and $\varepsilon^{(m)} w_0 \sim \varepsilon^{(m)} w'_0 \gamma$. We show that $(q_0, w) \vdash^*_{\mathfrak{A}} (q, \varepsilon)$ for some $w \in \Theta^*$ such that $\varepsilon^{(m)} w \sim \varepsilon^{(m)} w'_0 a\beta$. This will complete the proof because $w' = w'_0 a$.

Transition $\delta = (S', a, S)$ must have been added to Δ' in one of the two cases selected in Lines 12 or 14.

- Line 12: $\delta = (S', a, S)$ has been added to Δ' after S' had been selected as X in Line 6, a terminal $a_0 \in \Theta \setminus \mathcal{B}$ had been selected in Line 9, Y had been computed in Line 11, and $S = Y'$ had been identified by $Y = \beta' Y'$ for some blank $\beta' \in \mathcal{B}$ with $type(\beta') = (j, j)$ and $a \sim a_0\beta'$. Since $(\beta, q) \in S = Y'$, $Y = \beta' Y'$ must contain some (γ_0, q) with $\gamma_0 \sim \beta'\beta$. There must be a state $q'' \in Q$ such that $(\gamma_0, q) \in Cl(q'')$ and $q'' \in \psi(a_0)$, where ψ is computed in Line 8, because $(\gamma_0, q) \in Y$. By the definition of $Cl(X)$, $(\gamma_0, q) \in Cl(q'')$ implies $(q'', \sigma) \vdash^*_{\mathfrak{A}} (q, \varepsilon)$ for

some $\sigma \in \mathcal{B}^*$ such that $\varepsilon^{(j)}\sigma \sim \varepsilon^{(j)}\gamma_0$. By the definition of ψ, there must be a transition $(q', b, q'') \in \Delta$ and $(\gamma, q') \in S' = X$ such that $a_0 \sim \gamma b$ and thus $(q_0, w_0 b\sigma) \vdash^*_{\mathfrak{A}'} (q', b\sigma) \vdash_{\mathfrak{A}} (q'', \sigma) \vdash^*_{\mathfrak{A}} (q, \varepsilon)$ by the induction hypothesis for some $w_0 \in \Theta^*$ such that $\varepsilon^{(m)} w_0' \gamma \sim \varepsilon^{(m)} w_0$. Therefore,

$$\varepsilon^{(m)} w_0 b\sigma \sim \varepsilon^{(m)} w_0' \gamma b\sigma \sim \varepsilon^{(m)} w_0' a_0 \sigma \sim \varepsilon^{(m)} w_0' a_0 \gamma_0 \sim \varepsilon^{(m)} w_0' a_0 \beta' \beta \sim \varepsilon^{(m)} w_0' a\beta.$$

This proves the statement of the lemma in this case by choosing $w = w_0 b\sigma$.

- Line 14: $\delta = (S', a, S)$ has been added to Δ' after S' had been selected as X in Line 6 and $S = Y$ had been computed in Line 11 after selecting a as a_0 in Line 9. There must be a state $q'' \in Q$ such that $(\beta, q) \in Cl(q'')$ and $(a, q'') \in \psi$ computed in Line 8 because $(\beta, q) \in S = Y$. Similar to the previous case, $(\beta, q) \in Cl(q'')$ implies $(q'', \sigma) \vdash^*_{\mathfrak{A}} (q, \varepsilon)$ for some $\sigma \in \mathcal{B}^*$ such that $\varepsilon^{(j)}\sigma \sim \varepsilon^{(j)}\beta$. By the definition of ψ, there must be a transition $(q', b, q'') \in \Delta$ and $(\gamma, q') \in S' = X$ with $a \sim \gamma b$ and thus $(q_0, w_0 b\sigma) \vdash^*_{\mathfrak{A}} (q', b\sigma) \vdash_{\mathfrak{A}} (q'', \sigma) \vdash^*_{\mathfrak{A}} (q, \varepsilon)$ by the induction hypothesis for some $w_0 \in \Theta^*$ such that $\varepsilon^{(m)} w_0' \gamma \sim \varepsilon^{(m)} w_0$. Therefore,

$$\varepsilon^{(m)} w_0 b\sigma \sim \varepsilon^{(m)} w_0' \gamma b\sigma \sim \varepsilon^{(m)} w_0' a\sigma \sim \varepsilon^{(m)} w_0' a\beta.$$

This proves the statement in this case, and thus completes the proof of the lemma, by setting $w = w_0 b\sigma$. □

We are now ready to prove the equivalence of a finite automaton and its powerset automaton using the following notion of equivalence:

Definition 4.5 (Equivalent automata)**.** Two finite automata of the same type (m, n) are *equivalent* if the following holds: for every $w \in \mathcal{L}(\mathfrak{A})$ there exists $w' \in \mathcal{L}(\mathfrak{A}')$ such that $\varepsilon^{(m)} w \sim \varepsilon^{(m)} w'$, and for every $w' \in \mathcal{L}(\mathfrak{A}')$ there exists $w \in \mathcal{L}(\mathfrak{A})$ such that $\varepsilon^{(m)} w \sim \varepsilon^{(m)} w'$.

Theorem 4.6. *$\mathfrak{A}$ and the powerset automaton $\mathfrak{A}'$ computed from $\mathfrak{A}$ by Algorithm 1 are equivalent.*

Proof. We have to prove both implications in Definition 4.5. To prove the first implication, let $w \in \Theta^*$ and consider a sequence $(q_0, w) \vdash^*_{\mathfrak{A}} (q, \varepsilon)$ with $q \in F$. By Lemma 4.3, there exist $w'' \in \Theta^*$, $S \in Q'$, and $\beta \in \mathcal{B}$ such that $(S_0, w'') \vdash^*_{\mathfrak{A}'} (S, \varepsilon)$, $(\beta, q) \in S$, and $\varepsilon^{(m)} w \sim \varepsilon^{(m)} w''\beta$. Since $S \in Q'$, S must have been selected as X in Line 6 at some point. Since $q \in F$ and $(\beta, q) \in S$, a transition (S, β, S_f) has been added to Δ' in Line 7, and thus $(S_0, w''\beta) \vdash^*_{\mathfrak{A}'} (S, \beta) \vdash_{\mathfrak{A}'} (S_f, \varepsilon)$, i.e., $w''\beta \in \mathcal{L}(\mathfrak{A}')$, which proves the first implication by choosing $w' = w''\beta$.

Now consider any $w' \in \Theta^*$ and a sequence $(S_0, w') \vdash^*_{\mathfrak{A}'} (S_f, \varepsilon)$. Since S_f can only be reached by transitions added in Line 7, this sequence must have the form $(S_0, w''\beta) \vdash^*_{\mathfrak{A}'} (S, \beta) \vdash_{\mathfrak{A}'} (S_f, \varepsilon)$ for some $S \in q'$, $w'' \in \Theta^*$, $q \in F$, and $\beta \in \mathcal{B}$ such that $w' = w''\beta$ and $(\beta, q) \in S$. By Lemma 4.4, there is a $w \in \Theta^*$ such that $(q_0, w) \vdash^*_{\mathfrak{A}} (q, \varepsilon)$ and $\varepsilon^{(m)} w \sim \varepsilon^{(m)} w''\beta = \varepsilon^{(m)} w'$, which proves the other implication. □

4.2 Making the Powerset Automaton Deterministic

Since Algorithm 1 resembles the classical powerset construction, one might assume that the resulting powerset automaton is deterministic, but this is not always the case, demonstrated by the following example:

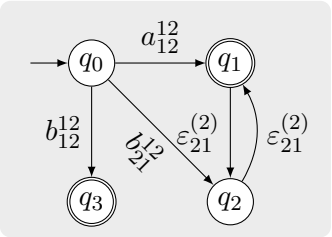


Figure 3: Automaton $\mathfrak{B}$ of Example 4.1.

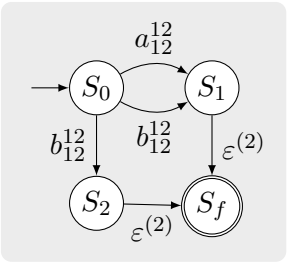


Figure 4: Nondeterministic automaton $\mathfrak{B}'$ produced from $\mathfrak{B}$ by Algorithm 1.

Example 4.1. Let a and b be symbols of rank 2, and consider the finite automaton $\mathfrak{B}$ shown in Figure 3 as input to Algorithm 1. Before the first iteration of the while loop beginning at Line 5 starts, we have $S_0 = \{(\varepsilon_{12}^{(2)}, q_0)\}$, $Q' = \{S_0, S_f\}$, $\Delta' = \varnothing$, and *frontier* $= \{S_0\}$. The first iteration of the while loop then selects $X = S_0$ and computes $\psi = \{(a_{12}^{12}, q_1), (b_{21}^{12}, q_2), (b_{12}^{12}, q_3)\}$. The nested foreach loop has the following iterations:

1. $a_0 = a_{12}^{12}$ and $Y = \{(\varepsilon_{12}^{(2)}, q_1), (\varepsilon_{21}^{(2)}, q_2)\}$. Lines 15 and 16 add Y as a new state S_1 to Q' and (S_0, a_{12}^{12}, S_1) to Δ'.
2. $a_0 = b_{21}^{12}$ and $Y = \{(\varepsilon_{12}^{(2)}, q_2), (\varepsilon_{21}^{(2)}, q_1)\}$. Since $Y = \varepsilon_{21}^{(2)} S_1$, Line 13 computes $a_1 = b_{12}^{12}$ because $b_{12}^{12} \sim b_{21}^{12}\varepsilon_{21}^{(2)}$, and adds (S_0, b_{12}^{12}, S_1) to Δ'.
3. $a_0 = b_{12}^{12}$ and $Y = \{(\varepsilon_{12}^{(2)}, q_3)\}$. Lines 15 and 16 add Y as a new state S_2 to Q' and (S_0, b_{12}^{12}, S_2) to Δ'.

Algorithm 1 terminates after adding $(S_1, \varepsilon_{12}^{(2)}, S_f)$ and $(S_2, \varepsilon_{12}^{(2)}, S_f)$ to Δ' with $Q' = \{S_0, S_1, S_2, S_f\}$. Figure 4 shows the resulting automaton $\mathfrak{B}'$. We have $(S_0, b_{12}^{12}, S_1), (S_0, b_{12}^{12}, S_2) \in \Delta'$, that is, $\mathfrak{B}'$ is nondeterministic. □

The reason for the nondeterminism of the resulting automaton is apparently the use of the atoms b_{12}^{12} and b_{21}^{12}, which is problematic because $b_{12}^{12} \sim b_{21}^{12}\varepsilon_{21}^{(2)}$. Let us call automata that use such symbols *ambiguous*:

Definition 4.7 (Ambiguous finite automata)**.** A finite automaton $\mathfrak{A} = (\Theta, Q, \Delta, q_0, F)$ is *ambiguous* if there are symbols $a, a' \in \Theta \setminus \mathcal{B}$ such that $a \neq a'$ and $a\beta \sim a'\beta'$ for some $\beta, \beta' \in \mathcal{B}$ (i.e., $[\![a]\!]$ and $[\![a']\!]$ differ only in their rears). $\mathfrak{A}$ is called *unambiguous* if it is not ambiguous.

Note that the automaton defined in Example 4.1 is ambiguous by this definition because $b_{21}^{12}\varepsilon_{21}^{(2)} \sim b_{12}^{12} \sim b_{12}^{12}\varepsilon_{12}^{(2)}$.

Theorem 4.8. *The powerset automaton $\mathfrak{A}'$ computed by Algorithm 1 is deterministic if $\mathfrak{A}$ is unambiguous.*

Proof. Consider any unambiguous automaton $\mathfrak{A}$, and assume that Algorithm 1 produces a nondeterministic automaton $\mathfrak{A}'$, i.e., there are two transitions $\delta = (S, a, S') \in \Delta'$ and $\delta' = (S, a, S'') \in \Delta'$ such that $S' \neq S''$. Note that we have target state S_f and $\beta \in \mathcal{B}$ for all transitions added to Δ' in Line 7, and $Y \neq S_f$, $Y' \neq S_f$, $a_0 \notin \mathcal{B}$, as well as $a_1 \notin \mathcal{B}$ for all transitions added in Lines 13 and 16, respectively. Consequently, neither δ nor δ' can have been added by Line 7, and both of them must have been added within the same iteration of the while loop using the same set ψ computed in Line 8, but in different iterations of the foreach loop that have selected, say, c_1 and

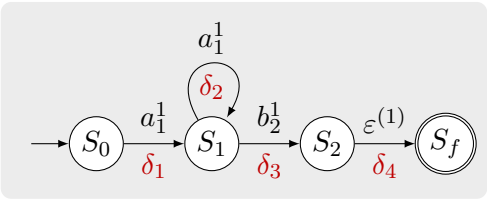


Figure 5: The deterministic automaton $\mathfrak{S}'$ obtained from $\mathfrak{S}$ (Figure 2) by Algorithm 1.

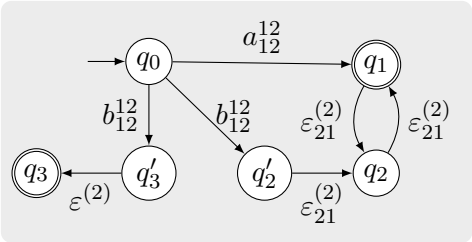


Figure 6: Unambiguous automaton $\mathfrak{B}''$ obtained from $\mathfrak{B}$ (Example 4.1).

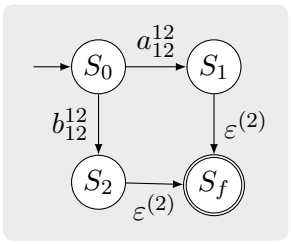


Figure 7: Automaton $\mathfrak{B}_\mathrm{d}$ obtained from $\mathfrak{B}''$ by Algorithm 1.

c_2 as a_0, respectively. If δ has been added to Δ' in Line 13, we have $a \sim c_1\beta_1$ for some $\beta_1 \in \mathcal{B}$, and $a = c_1$ if δ has been added in Line 16. Similarly, $a \sim c_2\beta_2$ for some $\beta_1 \in \mathcal{B}$, or $a = c_2$. We have $c_1 \neq c_2$ because c_1 and c_2 have been selected in different foreach loop iterations. Hence, either $c_1 \sim c_2\beta_2$, $c_1\beta_1 \sim c_2$, or $c_1\beta_1 \sim c_2\beta_2$, in contradiction to $\mathfrak{A}$ being unambiguous. □

Example 4.2. The automaton $\mathfrak{S}$ shown in Figure 2 is unambiguous, and Algorithm 1 thus computes a deterministic finite automaton (DFA) $\mathfrak{S}'$ shown in Figure 5. The states of $\mathfrak{S}'$ are $S_0 = \{(\varepsilon^{(1)}, q_0)\}$, $S_1 = \{(\varepsilon^{(1)}, q_0), (\varepsilon^{(1)}, q_1)\}$, $S_2 = \{\varepsilon^{(1)}, q_2)\}$, and $S_f = \varnothing$.

The restriction to unambiguous automata is in fact insignificant for automata over subsets of the canonical alphabet $\mathbf{\Theta}_\Sigma$ (and its canonical interpretation). One can easily transform an automaton $\mathfrak{A}$ into an equivalent unambiguous one using the following iterative process:

Consider an automaton $\mathfrak{A}$ that has $a, a' \in \Theta \setminus \mathcal{B}$ such that $a \neq a'$ and $a\beta \sim a'\beta'$ for some $\beta, \beta' \in \mathcal{B}$. Since we use the canonical interpretation of symbols in $\mathbf{\Theta}_\Sigma$, a and a' must have the form $a = b_\varrho^\varphi$ and $a' = b_{\varrho'}^\varphi$ using the same label $b \in \bar{\Sigma}$ and front interface φ, that is, a and a' differ only in their rear interfaces. Hence, one can identify a more general rear interface $\hat{\varrho}$ such that $a \sim b_{\hat{\varrho}}^\varphi \hat{\beta}$ and $a' \sim b_{\hat{\varrho}}^\varphi \hat{\beta}'$ for some blanks $\hat{\beta}$ and $\hat{\beta}'$. $\mathfrak{A}'$ is then obtained from $\mathfrak{A}$ by removing a and a' from Θ and adding $b_{\hat{\varrho}}^\varphi, \hat{\beta}$, and $\hat{\beta}'$. Furthermore, every transition (q, a, q') in Δ is replaced by two transitions $(q, b_{\hat{\varrho}}^\varphi, q'')$ and $(q'', \hat{\beta}, q')$ where q'' is a new state with $\mathit{rank}(q'') = |\hat{\varrho}|$ that is also added to Q, and proceeds similarly for each transition referring to a'. This process is continued until $\mathfrak{A}'$ is finally unambiguous.

Example 4.3. Figure 6 shows the unambiguous automaton $\mathfrak{B}''$ obtained from $\mathfrak{B}$ in Example 4.1 using the procedure sketched above. The ambiguous transitions (q_0, b_{21}^{12}, q_2) and (q_0, b_{12}^{12}, q_3) of $\mathfrak{B}$ have been split into four transitions with new intermediate states q_2' and q_3'. Algorithm 1 now produces the DFA $\mathfrak{B}_\mathrm{d}$ shown in Figure 7 when applied to $\mathfrak{B}''$.

4.3 Selecting Promising Transitions

Let us now consider the DFA $\mathfrak{A}_\mathrm{d}$ that has been produced by Algorithm 1, and use it for recognizing a valid graph $G \in \mathcal{L}_G(\mathfrak{A}_\mathrm{d})$ using graph moves. First note that $\mathfrak{A}_\mathrm{d}$ does not have any blank transitions except those that go into the final state, which does not have any outgoing transitions at all. Consequently, blank transitions are only applied when the remaining input is isomorphic to the corresponding blank graph, that is, there is no choice when such transitions have to be applied. Despite determinism, this is not necessarily the case for the other transitions. Consider the situation where we have reached a configuration (s,G) and G contains some edges. Although $\mathfrak{A}_\mathrm{d}$ is deterministic, several transitions may be applicable because a graph can be composed from basic graphs in different ways (see Examples 2.2 and 3.1). So we are again forced to choose and possibly to backtrack. To avoid this inefficient backtracking procedure, we would like to identify a unique transition, if it exists, that does not lead into a dead end, by inspecting only local information. To motivate such a procedure, let us first consider the simplest non-trivial case where s has two outgoing transitions, say $\delta,\delta' \in \Delta$ with $\delta \neq \delta'$. Now assume that we can prove that every edge read by a move using δ can never be read by any sequence of moves starting with δ'. It is then clear that any move using δ' instead of δ must inevitably lead into a dead end, and we know for sure that δ must be picked for continuing the recognition process. Moreover, backtracking later and trying δ' is meaningless because we already know that it will fail. It is thus crucial to find out whether an edge read by a move using δ can also be read by a sequence of moves starting with δ'. This is discussed next.

Again consider the situation where recognition has reached configuration (s,G), and two transitions $\delta,\delta' \in \Delta$, $\delta \neq \delta'$, can be used for two competing moves

$$(s,G) \Vdash_\delta (s',G') \tag{1}$$
$$(s,G) \Vdash_{\delta'} (s'',G'') \tag{2}$$

with $\delta = (s, a^{\varphi}_{\varrho}, s')$, $\delta' = (s, b^{\varphi'}_{\varrho'}, s'')$, $G = G_\delta \odot G' = G_{\delta'} \odot G''$ for two graphs G_δ and $G_{\delta'}$ with $\bar{G}_\delta = \{e\}$, $\bar{G}_{\delta'} = \{e'\}$, $G_\delta \cong [\![a^{\varphi}_{\varrho}]\!]$, and $G_{\delta'} \cong \left[\!\!\left[b^{\varphi'}_{\varrho'}\right]\!\!\right]$. The sequences φ and φ' specify how e and e' must be connected to front nodes of G such that (1) and (2) are a valid moves, respectively. To be more precise, let $\xi,\xi' \colon [\mathit{rank}(a)] \rightharpoonup [\mathit{rank}(s)]$ be partial functions defined by

$$\xi = \{(k,i) \in [\mathit{rank}(a)] \times [\mathit{rank}(s)] \mid k = \varphi(i)\}. \tag{3}$$
$$\xi' = \{(k,i) \in [\mathit{rank}(b)] \times [\mathit{rank}(s)] \mid k = \varphi'(i)\}. \tag{4}$$

Once more, we use the notation $\varphi(i)$ to access the i-th number within the sequence φ. Thus, ξ and ξ' assign the index of a front node (in the sequence of all front nodes of G) to its index in the sequence of all nodes attached to e or e', respectively. If a node is attached to e or e' as its k-th node, but is not a front node of G, $\xi(k)$ or $\xi'(k)$ are undefined, respectively.

We now assume that e is also read in the competing move (2) or a subsequent move. If e is read in (2), we have $e = e'$ as well as $a = b$, and $\xi = \xi'$ follows.

If $e \neq e'$, e must be read later, that is, there must be a sequence of moves

$$(s'',G'') \Vdash^*_{\mathfrak{A}_\mathrm{d}} (q,H) \Vdash_{\delta''} (q',H') \tag{5}$$

with $\delta'' = (q, a^{\varphi''}_{\varrho''}, q')$, $H = H_{\delta''} \odot H'$, and $\bar{H}_{\delta''} = \{e\}$, and $H_{\delta''} \cong \left[\!\!\left[a^{\varphi''}_{\varrho''}\right]\!\!\right]$.

Some front nodes of G may also be front nodes of H in (5). Let us indicate this situation by a partial function $\mu\colon [rank(s)] \rightharpoonup [rank(q)]$ defined by

$$\mu = \{(i,j) \in [rank(s)] \times [rank(q)] \mid front_G(i) = front_H(j)\} \tag{6}$$

Consequently, $\{front_G(i) \mid i \in dom(\mu)\}$ is the subset of those front nodes of G that are also front nodes of H. Similar to ξ and ξ', let us define the partial function $\xi''\colon [rank(a)] \rightharpoonup [rank(s)]$ by

$$\xi'' = \{(k,i) \in [rank(a)] \times dom(\mu) \mid k = \varphi''(\mu(i))\}. \tag{7}$$

Function μ is used to refer to front nodes of G instead of front nodes of H.

We now show that we have $\xi = \xi''$, similar to the case $e = e'$. To see this, consider any node v attached to e, that is, there is $k \in [rank(a)]$ such that $v = att_G(e,k)$.

- If v is not a front node of G, $k \notin [\varphi]$ follows from $G_\delta \cong [\![a^{\varphi}_{\varrho}]\!]$, and thus $k \notin dom(\xi)$. Now assume that $k \in dom(\xi'')$ and, hence, $k = \varphi''(j)$ as well as $j = \mu(i)$ for some $i \in [rank(s)]$ and $j \in [rank(q)]$. This implies $front_G(i) = front_H(j) = att_G(e,k) = v$ in contradiction to $v \notin [front_G]$. Hence, $k \notin dom(\xi'')$ as well.
- If v is a front node of G, there is an index $i \in [rank(s)]$ such that $v = front_G(i)$ and $k = \varphi(i)$, because of $G_\delta \cong [\![a^{\varphi}_{\varrho}]\!]$. Consequently, $k \in dom(\xi)$ and $i = \xi(k)$. When e is read in the last move (using δ'') in (5), it must also be a front node of H (otherwise, H could not contain v as a node because moves cannot turn front nodes into non-front nodes). Hence, $i \in dom(\mu)$ and $v = front_H(\mu(i))$. $H_{\delta''} \cong \left[\!\!\left[a^{\varphi''}_{\varrho''}\right]\!\!\right]$ also implies $k = \varphi''(\mu(i))$, and thus $k \in dom(\xi'')$ as well as $i = \xi''(k)$.

Note that the definition of ξ'' in fact subsumes the definition of ξ' when using the identity on $[rank(s)]$ as μ. Suppose we could compute the set $\Xi(\delta', a)$ of all functions ξ'' defined as in (7) for any graph G where μ represents any move sequence starting with δ' applied to G and finally reading an a-labelled edge (that is, exactly the situation as in (2) and (5)). Then we can summarize the discussion above as follows: If e can be read also in move (2) or later in (5), then $\xi \in \Xi(\delta', a)$. Since ξ and $\Xi(a)$ are independent of the specific choice of G and e, we can state the following observation:

Observation 4.9. *Let $\delta, \delta' \in \Delta$, ξ, and $\Xi(\delta', a)$ defined as described above, $\xi \notin \Xi(\delta', a)$, and $G \in \mathcal{G}_\Sigma$ any graph such that δ can be applied to G. Then δ' must not be tried for a move because it either cannot be applied to G, or its application leads inevitably into a dead end eventually.*

In fact, $\Xi(\delta', a)$ can be computed in a rather straightforward way using Algorithm 2. To simplify things, we use the notation $\Delta_q = \{\delta \in \Delta \mid \exists a \in \Theta, q' \in Q : \delta = (q, a, q')\}$ for indicating the set of transitions leaving a state q. Let us further define

$$follow(\delta) = \{(a, \xi) \mid \xi \in \Xi(\delta, a)\}. \tag{8}$$

The following lemma states that Algorithm 2 computes $follow(\delta)$.

Lemma 4.10. *Let Σ be a ranked alphabet, Θ a finite set of canonical graph symbols for Σ, $\mathfrak{A}_\mathrm{d} = (\Theta, Q, \Delta, q_0, F)$ a finite automaton produced by Algorithm 1, and $\delta_0 \in \Delta$ a transition. Called with $\mathfrak{A}_\mathrm{d}$ and δ_0, Algorithm 2 returns $follow(\delta_0)$ for $\mathfrak{A}_\mathrm{d}$.*

Proof Sketch. Consider any $\mathfrak{A}_\mathrm{d}$ and δ_0 as in the lemma. We first show that Algorithm 2 terminates. To see this, first note that $dom(\mu') \subseteq dom(\mu)$ and $\mu'(x) \in [rank(q')]$ for each $x \in dom(\mu')$

Algorithm 2: Determining $follow(\delta_0)$.

Input : Finite automaton $\mathfrak{A}_\mathrm{d} = (\Theta, Q, \Delta, q_0, F)$ produced by Algorithm 1 and a transition $\delta_0 \in \Delta$.

Output: Follow set *follow*.

```
1  let δ0 ∈ Δs
2  frontier ← {(δ0, ν)} where ν = {(i,i) | i ∈ [rank(s)]}
3  done ← ∅, follow ← ∅
4  while frontier ≠ ∅ do
5      select and remove any (δ, μ) from frontier
6      add (δ, μ) to done
7      let δ = (q, α, q′)
8      if α ∉ B then
9          let α = a_ϱ^φ
10         ξ = {(i,j) ∈ [rank(a)] × dom(μ) | i = φ(μ(j))}
11         add (a, ξ) to follow
12         μ′ ← {(i,j) ∈ dom(μ) × [rank(q′)] | φ(μ(i)) = ϱ(j)}
13         foreach δ′ ∈ Δ_q′ such that (δ′, μ′) ∉ done do
14             add (δ′, μ′) to frontier
```

of function μ' defined in Line 12, and thus $dom(\mu') \subseteq [rank(s)]$ for every pair (δ', μ') ever added to *frontier* in Line 14. Consequently, only finitely many pairs (δ, μ) can be added to *frontier* and *done*. And because no pair is added to *frontier* again after it has been selected in Line 5 (and adding it to *done* in Line 6), Algorithm 2 terminates.

Now consider any $a \in \Sigma$, $\xi \in \Xi(\delta_0, a)$ as for Observation 4.9, and any graph $G \in \mathbb{G}_\Sigma$. Consequently, there is a move sequence starting with δ_0 and finally reading an a-labeled edge using some transition δ such that ξ is defined like ξ' or ξ'' in (4) and (7), respectively, and using an appropriate partial function μ. By induction on the move sequence, one can show that *frontier* eventually will contain (δ, μ). Consequently, (a, ξ) is added to *follow* in Line 11, that is, (a, ξ) is contained in the result of Algorithm 2 as required.

For the other direction, consider any pair (a, ξ) in the result of Algorithm 2. By induction on the number of iterations of the while-loop starting at Line 4, one can show that the set *done* only contains pairs (δ, μ) such that there exists a move sequence starting with δ_0 and ending with δ so that μ is exactly as in (6). Since $(a, \xi) \in follow$, it must have been added to *follow* in Line 11 at some point after selecting (δ, μ) in Line 5. Using (4) and (7), one can show that $\xi \in \Xi(\delta_0, a)$ as required. □

We now use Observation 4.9 for identifying the unique transition (if it exists) that does not lead into a dead end, provided that $\mathfrak{A}_\mathrm{d}$ satisfies certain conditions. To this end, let us define

$$next(\delta) = (a, \xi) \tag{9}$$

for each transition $\delta = (s, a_\varrho^\varphi, s') \in \Delta$ where ξ is defined as in (3). Moreover, let $\prec_q \subseteq \Delta_q \times \Delta_q$ be such that

$$\delta \prec_q \delta' \quad \text{if} \quad next(\delta') \in follow(\delta) \tag{10}$$

for each $q \in Q$. According to Observation 4.9, $\delta' \not\prec_q \delta$ then indicates that δ' may only be tried if δ cannot be applied.

Consider any state $q \in Q$. If $\prec_q^+$ is irreflexive, one can extend $\prec_q$, by topological sorting, to a strict total order $\sqsubset_q \subseteq \Delta_q \times \Delta_q$ such that $\prec_q \subseteq \sqsubset_q$. Such an order $\sqsubset_q$ has the following nice property with respect to Observation 4.9:

Lemma 4.11. *Let $q \in Q$, $\sqsubset_q \subseteq \Delta_q \times \Delta_q$ a strict total order such that $\prec_q \subseteq \sqsubset_q$, and $\delta, \delta' \in \Delta_q$. Then $\delta \sqsubset_q \delta'$ implies $next(\delta) \notin follow(\delta')$.*

Proof. Consider any $q \in Q$ and $\delta, \delta' \in \Delta_q$ with $\delta \sqsubset_q \delta'$. Then $next(\delta) \in follow(\delta')$ would imply $\delta' \prec_q \delta$ and thus $\delta' \sqsubset_q \delta$, contradicting the irreflexivity of $\sqsubset_q$. □

Such a total strict order $\sqsubset_q$ only exists if $\prec_q^+$ is irreflexive, which motivates the following:

Definition 4.12 (Transition Selection Property)**.** A DFA $\mathfrak{A} = (\Theta, Q, \Delta, q_0, F)$ has the *transition selection* (*TS*) *property* if $\prec_q^+$ as defined in (10) is irreflexive for each state $q \in Q$.

An immediate consequence of Lemma 4.11 is the following:

Lemma 4.13. *For every DFA $\mathfrak{A}$ with the TS property, there is a constant time procedure which, given a configuration of $\mathfrak{A}$, selects the unique transition that allows to continue the recognition process without running into a dead end, provided that such a transition exists.*

Proof. Assume that a configuration with state q has been reached during recognition. There exists an order $\sqsubset_q$ as in Lemma 4.11 because of the TS property. Now try all outgoing transitions of q in ascending order of $\sqsubset_q$ and pick the first one, say δ, that can be applied. Then $\delta \sqsubset_q \delta'$ holds for every outgoing transition δ' that has not yet been tried, and thus $next(\delta) \notin follow(\delta')$ by Lemma 4.11, that is, δ' must not be tried by Observation 4.9 because δ can be applied.

The relation $\sqsubset_q$ depends only on $\mathfrak{A}$ and q, and can thus be precomputed for every state q. Checking whether a given transition applies takes constant time, because all that is required is to test whether an edge with the relevant label is attached to the front nodes of the graph as determined by the symbol to be read by the transition. Using appropriate data structures, this can straightforwardly be implemented to run in constant time. □

Example 4.4. Only state S_1 of the DFA $\mathfrak{S}'$ shown in Figure 5 has two outgoing transitions, δ_2 and δ_3. We obtain

$$next(\delta_2) = (a, \{(1,1)\}) \qquad follow(\delta_2) = \{(a, \{(1,1)\}), (b, \{(1,1)\})\}$$
$$next(\delta_3) = (b, \{(1,1)\}) \qquad follow(\delta_3) = \{(b, \{(1,1)\})\}$$

by applying (9) with (3) and using Algorithm 2, that is, $\delta_2 \prec_{S_1} \delta_3$ and $\delta_3 \not\prec_{S_1} \delta_2$, that is, $\mathfrak{S}'$ has the TS property. We can thus choose $\delta_2 \sqsubset_{S_1} \delta_3$, that is, one must try δ_2 first when one has reached S_1, and δ_3 only if δ_2 is not applicable, that is, if there is no remaining outgoing edge with label a.

4.4 Free Edge Choice

Unfortunately, the transition selection property does not necessarily prevent the recognition process from running into dead ends, even for valid graphs. This is the case if one has the choice between different edges to be read by a selected transition, but not all of them are equally suited. This is demonstrated in the following example:

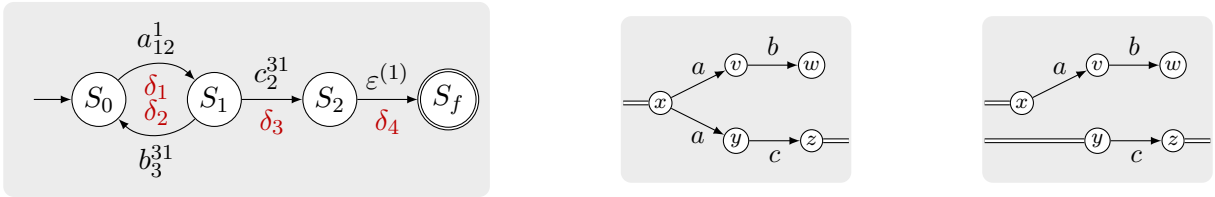


Figure 8: Finite automaton $\mathfrak{F}$ and two graphs used in Example 4.5.

Example 4.5. Let a, b, and c be symbols of rank 2. The DFA $\mathfrak{F}$ shown in Figure 8 always allows to select a unique transition: δ_1 in state S_0, δ_2 in S_1 if the second front interface node has an outgoing b-edge, and δ_3 if it has an outgoing c-edge. The left graph in Figure 8 can thus be recognized by first reading the edge from x to v in state S_0 and then following the uniquely determined moves.

However, the recognition process will run into a dead end if it reads the edge from x to y first. This move then yields a configuration (S_1, G') where G' is the right graph in Figure 8. The second front interface node (y) has an outgoing c-edge, but no b-edge, that is, δ_3 is selected. This transition tries to find a composition $G' = C \odot G''$ with $C \cong [\![c^{31}_2]\!]$, that is, $\mathit{front}_C = xy$, $\mathit{rear}_C = z$, and thus $x \notin \dot{G}''$, which would leave the edge from x to v dangling. Hence, δ_3 cannot be applied which leaves the recognition process stuck in a dead end. □

Such a situation cannot happen if every edge that can be chosen to be read next allows to continue the recognition process until an accepting configuration is reached, or none of them does. We say that the next edge can be freely chosen:

Definition 4.14 (Free Edge Choice Property)**.** A finite automaton $\mathfrak{A} = (\Theta, Q, \Delta, q_0, F)$ has the *free edge choice (FEC) property* if $(q', H') \in \mathcal{C}_\mathfrak{A}$ implies $(q', H'') \in \mathcal{C}_\mathfrak{A}$ for all move sequences $(q_0, G) \Vdash^*_\mathfrak{A} (q, H) \vdash_\delta (q', H')$ and $(q, H) \vdash_\delta (q', H'')$ where $\delta \in \Delta$.[3]

By the discussion above, the recognition process using an automaton with the TS property and the FEC property will never run into a dead end when applied to a valid graph. It remains to show how the FEC property can be checked for a given automaton. We present a sufficient condition that is easy to test. Thus, an automaton is guaranteed to have the FEC property if it passes the test, but the converse is not necessarily true.

Consider a situation during the recognition of a valid graph when a transition has been selected for the next move, and there are several edges one can choose from. All the edges that are not being chosen now must be read later during the recognition process. This is only possible for certain transitions. Let us call these transitions *deferrable* because one edge can be read now whereas reading of the others is deferred:

A transition $\delta \in \Delta$ is called *deferrable* if and only if there are move sequences

$$(q, G) \vdash_\delta (q', G') \tag{11}$$

$$\text{and} \quad (q, G) \vdash_\delta (q', G'') \Vdash^*_\mathfrak{A} (s, H) \vdash_\mathfrak{A} (s', H') \tag{12}$$

such that the edge read in (11) and the one read in the last move of (12) are the same, that is $\bar{G}' \setminus \bar{G} = \bar{H}' \setminus \bar{H}$.

[3] Recall that $\mathcal{C}_\mathfrak{A}$ is the set of all acceptable graph configurations of $\mathfrak{A}$, that is, those configurations from where an accepting configuration can be reached (see Definition 3.3).

One can easily check with an algorithm similar to Algorithm 2 whether a transition is deferrable. With this information, one can then check whether $\mathfrak{A}$ has the FEC property:

Lemma 4.15. *A finite automaton $\mathfrak{A}$ has the free edge choice property if $[\varrho] \subseteq [\varphi]$ holds for every transition $(q, a^{\varphi}_{\varrho}, q')$ of $\mathfrak{A}$ that is deferrable.*

Proof. Consider a finite automaton $\mathfrak{A} = (\Theta, Q, \Delta, q_0, F)$ that satisfies the condition in the lemma. Let us assume that $\mathfrak{A}$ does not have the free edge choice property, that is, there are move sequences $(q_0, G) \vdash^*_{\mathfrak{A}} (q, H) \vdash_{\delta} (q', H') \in \mathcal{C}_{\mathfrak{A}}$ and $(q, H) \vdash_{\delta} (q', H'') \notin \mathcal{C}_{\mathfrak{A}}$. The two moves using δ must have read two different edges, say e and e', that is, $H = F \odot H' = F' \odot H''$, $\bar{F} = \{e\}$, and $\bar{F}' = \{e'\}$. H' then still contains e', which is read later, because $(q', H') \in \mathcal{C}_{\mathfrak{A}}$. Consequently, there is a move sequence $(q', H') \vdash^*_{\mathfrak{A}} (s, I) \vdash_{\mathfrak{A}} (s', I') \in \mathcal{C}_{\mathfrak{A}}$ such that $\bar{I} \setminus \bar{I}' = \{e'\}$, that is, δ is deferrable. Now let $\delta = (q, a^{\varphi}_{\varrho}, q')$. We have $a = lab_G(e) = lab_G(e')$ since e and e' can both be read by δ. And we have $[\varrho] \subseteq [\varphi]$ since δ is deferrable, and $\mathfrak{A}$ satisfies the condition in the lemma. Note that $front_H = front_F = front_{F'}$. Thus $[\varrho] \subseteq [\varphi]$ implies $rear_F = rear_{F'} = front_{H'} = front_{H''}$. Consequently, all nodes of H that are attached to e or e', but that are not in $front_H$, cannot be attached to any other edge, and thus $H' \cong H''$, where the isomorphism maps e to e' (and vice versa). Then $(q', H'') \in \mathcal{C}_{\mathfrak{A}}$ follows from $(q', H') \in \mathcal{C}_{\mathfrak{A}}$, contradicting the assumption. $\mathfrak{A}$ hence has the free edge property. □

Example 4.6. Transition δ_1 is the only deferrable transition of automaton $\mathfrak{F}$ in Figure 8, but it violates the condition of Lemma 4.15 as it reads a^1_{12}, and $[\varrho] = \{1, 2\} \not\subseteq \{1\} = [\varphi]$. Consequently, $\mathfrak{F}$ does not pass the FEC test, which leaves the question open whether $\mathfrak{F}$ has the FEC property or not. In fact, we have already seen in Example 4.5 that $\mathfrak{F}$ does not have the FEC property.

The automaton $\mathfrak{S}'$ in Figure 5, however, has the FEC property according to Lemma 4.15 since δ_2 is its only deferrable transition, and $[\varrho] = \{1\} \subseteq \{1\} = [\varphi]$ for the graph symbol a^1_1 read by δ_2. Since $\mathfrak{S}'$ also has the TS property (see Example 4.4), one can use $\mathfrak{S}'$ for recognizing star graphs without any backtracking.

4.5 Results

The following theorem summarizes the findings of this paper.

Theorem 4.16. *1. Every finite automaton over the canonical alphabet can effectively be turned into a DFA.*[4]

2. *Given a DFA $\mathfrak{A}$ (of the type produced by Algorithm 1), it can be decided in polynomial time (in the number of transitions of $\mathfrak{A}$) whether $\mathfrak{A}$ has the TS and the FEC property.*
3. *If $\mathfrak{A}$ has the TS and the FEC property, the membership problem for $\mathcal{L}(\mathfrak{A})$ can be decided in linear time.*

Proof. The first statement follows from Theorem 4.8 and the previously mentioned fact that ambiguous automata can be made unambiguous. Deciding the TS property requires polynomial time because Algorithm 2 and topological sorting must be used for checking irreflexivity of $\prec^+_q$ for every state q of $\mathfrak{A}$. The second statement then follows from Lemma 4.15 and the fact that it is decidable in polynomial time whether a transition is deferrable (see above). Finally, to decide whether a graph belongs to $\mathcal{L}(\mathfrak{A})$, we can repeatedly apply transitions using Lemma 4.13. □

[4] As usual, and unavoidably, the powerset construction can cause an exponential blow-up in the size of the automaton.

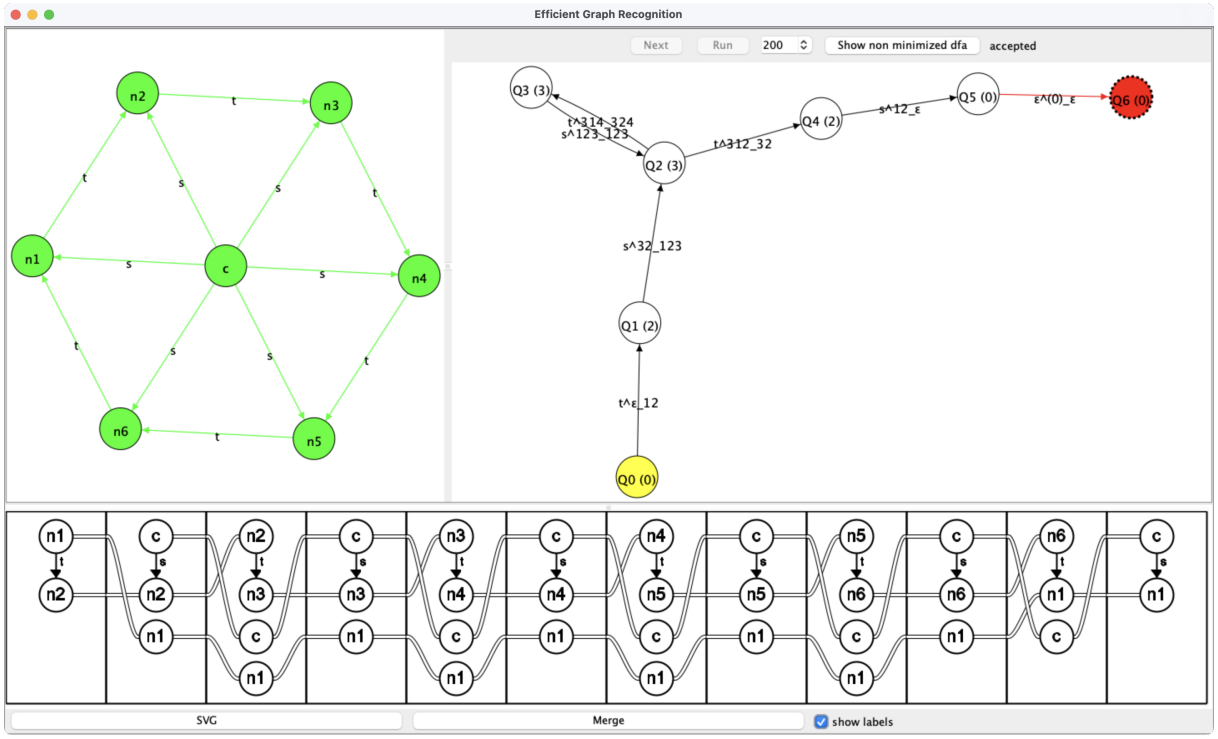


Figure 9: Screenshot of a tool realizing efficient graph recognition based on finite automata.

The concepts described in this paper have been realized in a graphical tool. It reads in a finite automaton over the canonical alphabet, or creates it from a regular expression in the obvious way, turns it into a DFA, and uses it for recognizing input graphs if the DFA has the TS and the FEC property. Figure 9 shows a screenshot with an input graph in the top-left window, and the DFA in the top-right window after the DFA has recognized the input graph. The bottom window shows how the input graph has been composed from basic graphs such that it has been accepted by the DFA.

Table 1 shows some example graph languages that can be specified and efficiently recognized by finite automata: *Stars* is the language of stars used in this paper (Example 3.1), *Wheels* contains wheel graphs introduced in [23, p. 92], *Palindromes* contains all palindromes over $\{a,b\}$, and $a^nb^nc^n$ the language $\{a^nb^nc^n \mid n > 0\}$ as string graphs [23, Ex. 2.4]. Table 1 shows the type of the contained graphs, that is, the size of their front and rear interfaces, and the size of their DFA in terms of the number of states and transitions. Moreover, *max. #nodes* shows the number of nodes of the largest atom used in the DFA. Each of the automata has the TS and FEC property.

Figure 9 shows the DFA for wheels and a wheel with six spokes (`s`-edges for spokes and `t`-edges for the tread of the wheel). The front and rear interfaces of the wheel are empty. As shown in the bottom window, the left-most atom refers to the `t`-edge from `n1` to `n2`, and recognition of the wheel has started with this part of the tread of the wheel. Recognition could have started with any of the `t`-edges because the DFA has the FEC property. However, the FEC test fails for this DFA because of the first transition from `Q0` to `Q1` with label `t^ε_12`, which represents t^{ε}_{12}. This transition is deferrable, yet $\{1,2\} \not\subseteq \varnothing$. Hence, *Wheels* is an example where the simple

Table 1: Example graph languages and their finite automata.

Language	Type	#states	#transitions	max. #nodes	TS	FEC
Stars	$(1,1)$	4	4	2	✓	✓
Wheels	$(1,0)$	7	7	4	✓	✓
Palindromes	$(2,0)$	5	7	3	✓	✓
$a^n b^n c^n$	$(2,0)$	9	10	4	✓	✓

FEC test fails although the automaton does have the FEC property.

5 Conclusions

In this paper we have defined finite automata for graph recognition, devised a modified powerset construction to make them deterministic, and stated two criteria (transition selection and free edge choice) under which these automata work without backtracking.

This paper leaves some questions open. Do (nondeterministic) finite graph automata recognize NP-complete languages, which exist for (context-free) hyperedge replacement grammars? How do the languages accepted by finite graph automata relate to those definable with the "regular" graph grammars of Gilroy *et al.* [20], which have an efficient parsing algorithm (not defined with automata) that also requires free edge choice? (Probably these language classes are incomparable.)

We intend to continue our work in several directions. First, it is easy to define graph expressions defining graph languages by composition, Kleene star, and union. These expressions could be used in the nested graph conditions of Habel and Pennemann [24, 30] to specify global graph properties [19], and check them with automata. Second, we could consider borrowed nodes in graph expressions and graph automata that can be contracted with other nodes in a graph [16]. Then these mechanisms allow to define graphs of unbounded treewidth.

Acknowledgments. We are very grateful to the anonymous reviewers for their helpful comments and suggestions, which have helped to improve the quality of this paper.

References

[1] IJ.J. Aalbersberg, Andrzej Ehrenfeucht & Grzegorz Rozenberg (1986): *On the Membership Problem for Regular DNLC Grammars. Discrete Applied Mathematics* 13, pp. 79–85, doi:10.1016/0166-218X(86)90070-3.

[2] Michael A. Arbib & Yehoshafat Give'on (1968): *Algebra Automata I: Parallel Programming as a Prolegomena to the Categorical Approach. Information and Control* 12(4), pp. 331–345, doi:10.1016/S0019-9958(68)90374-4.

[3] Christoph Blume, H.J. Sander Bruggink, Martin Friedrich & Barbara König (2013): *Treewidth, Pathwidth and Cospan Decompositions with Applications to Graph-Accepting Tree Automata. Journal of Visual Languages & Computing* 24(3), pp. 192–206, doi:10.1016/j.jvlc.2012.10.002.

[4] Symeon Bozapalidis & Antonios Kalampakas (2006): *Recognizability of graph and pattern languages. Acta Informatica* 42(8-9), pp. 553–581, doi:10.1007/s00236-006-0006-z.

[5] Symeon Bozapalidis & Antonios Kalampakas (2008): *Graph automata*. *Theoretical Computer Science* 393(1-3), pp. 147–165, doi:10.1016/j.tcs.2007.11.022.

[6] Franz-Josef Brandenburg & Konstantin Skodinis (2005): *Finite graph automata for linear and boundary graph languages*. *Theoretical Computer Science* 332(1-3), pp. 199–232, doi:10.1016/j.tcs.2004.09.040.

[7] H. J. Sander Bruggink & Barbara König (2018): *Recognizable languages of arrows and cospans*. *Math. Struct. Comput. Sci.* 28(8), pp. 1290–1332, doi:10.1017/S096012951800018X.

[8] David Chiang, Jacob Andreas, Daniel Bauer, Karl Moritz Hermann, Bevan Jones & Kevin Knight (2013): *Parsing Graphs with Hyperedge Replacement Grammars*. In: *Proc. 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Association for Computational Linguistics, Sofia, Bulgaria, pp. 924–932. Available at https://aclanthology.org/P13-1091.

[9] Bruno Courcelle (1990): *The Monadic Second-Order Logic of Graphs. I. Recognizable Sets of Finite Graphs*. *Information and Computation* 85(1), pp. 12–75, doi:10.1016/0890-5401(90)90043-H.

[10] Frank Drewes (1993): *Recognising k–connected hypergraphs in cubic time*. *Theoretical Computer Science* 109, pp. 83–122, doi:10.1016/0304-3975(93)90065-2.

[11] Frank Drewes, Berthold Hoffmann & Mark Minas (2015): *Predictive Top-Down Parsing for Hyperedge Replacement Grammars*. In Francesco Parisi-Presicce & Bernhard Westfechtel, editors: *Graph Transformation - 8th International Conf., ICGT 2015. Proceedings*, *LNCS* 9151, Springer, pp. 19–34, doi:10.1007/978-3-319-21145-9_2.

[12] Frank Drewes, Berthold Hoffmann & Mark Minas (2017): *Predictive Shift-Reduce Parsing for Hyperedge Replacement Grammars*. In Fabio Gadducci & Timo Kehrer, editors: *Graph Transformation - 10th International Conf., ICGT 2017, Proceedings*, *LNCS* 12741, Springer, pp. 106–122, doi:10.1007/978-3-319-61470-0_7.

[13] Frank Drewes, Berthold Hoffmann & Mark Minas (2019): *Extending Predictive Shift-Reduce Parsing to Contextual Hyperedge Replacement Grammars*. In Esther Guerra & Fernando Orejas, editors: *Graph Transformation - 12th International Conf., ICGT 2019*, *LNCS* 11629, Springer, pp. 55–72, doi:10.1007/978-3-030-23611-3_4.

[14] Frank Drewes, Berthold Hoffmann & Mark Minas (2019): *Formalization and Correctness of Predictive Shift-Reduce Parsers for Graph Grammars based on Hyperedge Replacement*. *Journal on Logical and Algebraic Methods for Programming* 104, pp. 303–341, doi:10.1016/j.jlamp.2018.12.006.

[15] Frank Drewes, Berthold Hoffmann & Mark Minas (2021): *Rule-Based Top-Down Parsing for Acyclic Contextual Hyperedge Replacement Grammars*. In Fabio Gadducci & Timo Kehrer, editors: *Graph Transformation - 14th International Conference, ICGT 2021*, *LNCS* 12741, Springer, pp. 164–184, doi:10.1007/978-3-030-78946-6_9.

[16] Frank Drewes, Berthold Hoffmann & Mark Minas (2022): *Acyclic Contextual Hyperedge Replacement: Decidability of Acyclicity and Generative Power*. In Nicolas Behr & Daniel Strüber, editors: *Graph Transformation - 15th International Conference, ICGT 2022*, *LNCS* 13349, Springer, pp. 3–19, doi:10.1007/978-3-031-09843-7_1.

[17] Joost Engelfriet & Jan Joris Vereijken (1997): *Context-Free Graph Grammars and Concatenation of Graphs*. *Acta Informatica* 34(10), pp. 773–803, doi:10.1007/s002360050106.

[18] Fabio Gadducci & Reiko Heckel (1997): *An inductive view of graph transformation*. In Francesco Parisi-Presicce, editor: *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97*, *LNCS* 1376, Springer, pp. 223–237, doi:10.1007/3-540-64299-4_36.

[19] Haim Gaifman (1982): *On local and non-local properties*. In J. Stern, editor: *Proc. of the Herbrand Symposium, Logic Colloquium*, *Studies in Logic and the Foundations of Mathematics* 105, North-Holland, pp. 105–135, doi:10.1016/S0049-237X(08)71879-2.

[20] Sorcha Gilroy, Adam Lopez & Sebastian Maneth (2017): *Parsing Graphs with Regular Graph Grammars*. In: *Proceedings of the 6th Joint Conference on Lexical and Computational Semantics (*SEM 2017)*, Association for Computational Linguistics, Vancouver, Canada, pp. 199–208, doi:10.18653/v1/S17-1024.

[21] Sorcha Gilroy, Adam Lopez, Sebastian Maneth & Pijus Simonaitis (2017): *(Re)introducing Regular Graph Languages*. In Makoto Kanazawa, Philippe de Groote & Mehrnoosh Sadrzadeh, editors: *Proceedings of the 15th Meeting on the Mathematics of Language, MOL 2017, London, UK, July 13-14, 2017*, ACL, pp. 100–113, doi:10.18653/v1/w17-3410.

[22] Yehoshafat Give'on & Michael A. Arbib (1968): *Algebra Automata II: The Categorical Framework for Dynamic Analysis*. *Information and Control* 12(4), pp. 346–370, doi:10.1016/S0019-9958(68)90381-1.

[23] Annegret Habel (1992): *Hyperedge Replacement: Grammars and Languages*. *LNCS* 643, Springer, doi:10.1007/BFb0013875.

[24] Annegret Habel & Karl-Heinz Pennemann (2005): *Nested Constraints and Application Conditions for High-Level Structures*. In H.-J. Kreowski et al., editors: *Formal Methods in Software and System Modeling*, *LNCS* 3393, Springer, pp. 293–308, doi:10.1007/978-3-540-31847-7_17.

[25] Berthold Hoffmann & Mark Minas (2017): *Generating Efficient Predictive Shift-Reduce Parsers for Hyperedge Replacement Grammars*. In M. Seidl & S. Zschaler, editors: *STAF 2017 Workshops*, *LNCS* 10748, Springer, pp. 76–91, doi:10.1007/978-3-319-74730-9_7.

[26] Antonios Kalampakas (2011): *Graph Automata: The Algebraic Properties of Abelian Relational Graphoids*. In Werner Kuich & George Rahonis, editors: *Algebraic Foundations in Computer Science - Essays Dedicated to Symeon Bozapalidis on the Occasion of His Retirement*, *LNCS* 7020, Springer, pp. 168–182, doi:10.1007/978-3-642-24897-9_8.

[27] Michael Kaminski & Shlomit S. Pinter (1992): *Finite Automata on Directed Graphs*. *Journal of Computer and System Sciences* 44(3), pp. 425–446, doi:10.1016/0022-0000(92)90012-8.

[28] Klaus-Jörn Lange & Emo Welzl (1987): *String Grammars with Disconnecting or a Basic Root of the Difficulty in Graph Grammar Parsing*. *Discrete Applied Mathematics* 16, pp. 17–30, doi:10.1016/0166-218X(87)90051-5.

[29] Clemens Lautemann (1990): *The complexity of graph languages generated by hyperedge replacement*. *Acta Informatica* 27, pp. 399–421, doi:10.1007/BF00289017.

[30] Karl-Heinz Pennemann (2009): *Development of Correct Graph Transformation Systems*. Dissertation, Carl-von-Ossietzky-Universität Oldenburg. Available at http://oops.uni-oldenburg.de/884/1/pendev09.pdf.

[31] Walter Vogler (1991): *Recognizing Edge Replacement Graph Languages in Cubic Time*. In H. Ehrig, H.-J. Kreowski & G. Rozenberg, editors: *Proc. Fourth Intl. Workshop on Graph Grammars and Their Application to Comp. Sci.*, *LNCS* 532, Springer, pp. 676–687, doi:10.1007/BFb0017421.

[32] Kurt-Ulrich Witt (1981): *Finite Graph-Acceptors and Regular Graph-Languages*. *Information and Control* 50(3), pp. 242–258, doi:10.1016/S0019-9958(81)90351-X.

Model-Driven Rapid Prototyping for Control Algorithms with the GIPS Framework (System Description)

Maximilian Kratz
Technical University of Darmstadt
Real-Time Systems Lab
Darmstadt, Germany
maximilian.kratz@es.tu-darmstadt.de

Sebastian Ehmes
Technical University of Darmstadt
Real-Time Systems Lab
Darmstadt, Germany
sebastian.ehmes@es.tu-darmstadt.de

Philipp Maximilian Menzel
Technical University of Darmstadt
Darmstadt, Germany
philipp_maximilian.menzel
@stud.tu-darmstadt.de

Andy Schürr
Technical University of Darmstadt
Real-Time Systems Lab
Darmstadt, Germany
andy.schuerr@es.tu-darmstadt.de

Software engineers are faced with the challenge of creating control algorithms for increasingly complex dynamic systems, such as the management of communication network topologies. To support rapid prototyping for these increasingly complex software systems, we have created the GIPS (**G**raph-Based **I**LP **P**roblem **S**pecification) framework[1] to derive some or even all of the building blocks of said systems, by using Model-Driven Software Engineering (MDSE) approaches. Developers can use our high-level specification language GIPSL (**G**raph-Based **I**LP **P**roblem **S**pecification **L**anguage) to specify their desired model optimization as sets of constraints and objectives. GIPS is able to derive executable (Java) software artifacts automatically that optimize a given input graph instance at runtime, according to the specification. Said artifacts can then be used as system blocks of, e.g., topology control systems. In this paper, we present the maintenance of (centralized) tree-based peer-to-peer data distribution topologies as a possible application scenario for GIPS in the topology control domain. The presented example is implemented using open-source software and its source code as well as an executable demonstrator in the form of a virtual machine is available on GitHub.

1 Introduction

Software developers working on modern (self-adaptive) control algorithms must deal with the ever-growing complexity of these types of software systems [8]. In the software engineering domain, model-driven software engineering tools have long been established as a valid approach for tackling the challenges of growing software complexity. Consequently, one could use the principles of Model-Driven Software Engineering (MDSE) and apply them to the development of control algorithms in general. Hence, we propose to support the process of developing control algorithms with our newly developed framework GIPS (**G**raph-Based **I**LP **P**roblem **S**pecification Tool) [12], which lends itself to a rapid prototyping approach following the idea of models@run.time [5, 6, 11]. GIPS embodies the MDSE approach by automatically deriving (Java) runtime artifacts from a given high-level specification, using our Domain-Specific Language (DSL) GIPSL (**G**raph-Based **I**LP **P**roblem **S**pecification **L**anguage) that is able to specify Graph Transformation (GT) rules. It uses typed and attributed graphs as input as well as output models and performs model transformations based on the formally founded and established GT

[1] GIPS - https://gips.dev

J. Endrullis, D. Grzelak, T. Heindel and J. Kosiol (Eds):
GCM 2023 and GCM 2024
EPTCS 417, 2025, pp. 157–172, doi:10.4204/EPTCS.417.9

framework [13]. In combination with the well-known Integer Linear Programming (ILP) optimization approach [7], GIPS can be used to obtain sets of model transformations that adhere to global and local constraints as well as deliver results that optimize a given cost function. Since GIPS operates on generic structures such as graphs and uses a generic optimization approach, it is not limited to a specific problem domain. In this paper, we make use of the generic nature of GIPS and present its application in the prototype development of an incremental algorithm that maintains (centralized) tree-based Peer-To-Peer (P2P) overlay networks for data distribution in a video streaming scenario, with the goal to reduce server load and increase robustness. The idea is to build this algorithm as a control algorithm in the form of the well-known MAPE-K loop [8, 18], where most of the system building blocks are automatically generated from a high-level specification by GIPS. In contrast to [12], which showed a GIPS-based application in the domain of data center networks, this paper presents GIPS as a means to facilitate model-driven rapid prototyping and gives some thoughts on how to use the results of the prototyping phase regarding (topology) control algorithms using an example.

The rest of the paper is structured as follows. In Section 2.1 and Section 2.2, we discuss the necessary basics of ILP and GT to give some fundamental knowledge of the underlying technologies of GIPS. Section 3 gives a rough overview of our new framework GIPS. In Section 4, we introduce the aforementioned example scenario of the P2P document distribution for the streaming platform lectureStudio[2] and our MAPE-K-based solution to the problem. This section is split into the problem description (Section 4.1), a possible solution approach (Section 4.2) together with a demonstration (Section 4.3), and a discussion on further challenges in the development process (Section 4.4). In Section 5, we discuss related work. Finally, Section 6 sums up our contribution and gives possible future enhancements.

The example implementation and the demonstrator in the form of a virtual machine are publicly available on GitHub[3].

2 Preliminaries

2.1 Integer Linear Programming

Integer Linear Programming (ILP) is an optimization approach that can be used to find the minimum (maximum) of an objective function $F : \mathbb{Z}^n \to \mathbb{R}$ by solving for an integer target vector $\vec{x} \in \mathbb{Z}^n$ while adhering to some constraints $f_j(\vec{x}) \leq 0$ $(j = 1, \ldots, m)$ [3, 7, 19]. Eq. (1) shows its canonical form, where $\vec{b} \in \mathbb{R}^m$ and $\vec{c} \in \mathbb{R}^n$ are vectors, $A \in \mathbb{R}^{m \times n}$ is a coefficient matrix, and $\vec{x} \in \mathbb{Z}^n$ is the solution as a vector.

$$\text{minimize } \vec{c}^T \vec{x} \text{ s. t. } A\vec{x} \leq \vec{b}, \ \vec{x} \geq 0, \text{ and } \vec{x} \in \mathbb{Z}^n \tag{1}$$

The goal of the optimization (minimization or maximization) can be converted to the respective counterpart if the objective function is multiplied by the factor -1. If all entries of the vector $\vec{x}$ have to be either 0 or 1, i.e., they are Boolean variables, the ILP problem is called a bivalent linear problem [7]. Tomaszek et al. [21] showed that the network embedding problem can be encoded as a bivalent linear problem. To achieve this problem formulation, we have to model the problem as a set of unknown integer variables, linear constraints, and a linear target function. The problem can afterwards be solved as an integer linear problem using solvers like Gurobi[4] (commercial) or GLPK[5] (free and open-source software).

[2] lectureStudio - `https://www.lecturestudio.org`

[3] GIPS GCM 2023 Artifact VM - `https://github.com/Echtzeitsysteme/gips-gcm-2023-artifact-vm`

[4] Gurobi Optimizer - `https://www.gurobi.com/solutions/gurobi-optimizer/`

[5] GNU Linear Programming Kit - `https://www.gnu.org/software/glpk/`

2.2 Graph Transformation

GIPS operates on models based on typed and attributed graphs, where objects correspond to typed nodes and references between objects correspond to typed edges. For this reason, we can make use of GTs, which is a formal framework that provides a rule-based way to define model transformations on graphs. GT rules consist of a Left-Hand Side (LHS) and a Right-Hand Side (RHS). Both, the LHS and the RHS are graph patterns that describe together which structures must be present, absent, created, or deleted in a given target graph. The LHS defines which graph structures have to be present in a target graph before a GT rule can be applied. The RHS, on the other hand, defines which graph structures must be present in the target graph after the rule has been applied. Unsurprisingly, GT heavily depends on graph Pattern Matching (PM), which is used to find subgraphs in a given target graph that match a given graph pattern, e.g., the LHS of a GT rule. Such a subgraph is called a match and consists of graph pattern nodes successfully mapped to a set of target graph nodes. GIPS relies on Incremental Graph Pattern Matching (IGPM), a commonly used PM approach, which keeps track of individual model changes, to update sets of appearing or vanishing matches incrementally. In GIPS, we make use of the HiPE[6] pattern matching engine, which is based on a massively parallelized variant of Forgy's Rete-approach [15].

3 The GIPS Framework

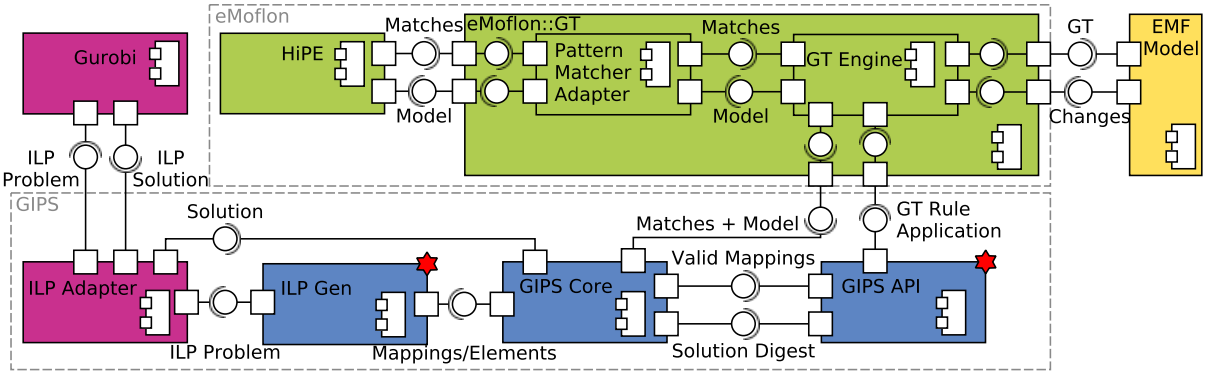


Figure 1: Component diagram of the GIPS framework (red star = generated during build time) [12].

The two driving motivations behind GIPS are: (1) Reducing the amount of effort (i.e., code) that has to be put into the creation of complex prototypes of graph optimization tools and (2) a reduction of programming errors that will inevitably occur in sufficiently complex projects, regardless of the programmer's skills. For this purpose, we have developed GIPSL that enables expressing ILP constraints in a UML/OCL-like fashion and which extends the eMoflon::IBeX-GT language[7] that provides the ability to specify GT rules and graph patterns. As a result, we gain a level of integration that allows constraint definitions to access and, thus, make use of graph nodes of GT rules or graph patterns. In essence, an output model of GIPS is shaped by a set of GT rule applications, which describe local modifications to a given model and are only executed if a rule's LHS holds beforehand. A subset of valid matches of a specific GT rule's LHS will be determined by the ILP solver. Said matches are only valid if they adhere to all constraints defined in the corresponding GIPSL specifications. By imposing ILP constraints

[6]HiPE - `https://github.com/HiPE-DevOps/HiPE-Updatesite`
[7]GIPS uses eMoflon::IBeX as a GT engine - `https://emoflon.org/ibex`

that interweave a set of GT rule applications, we can enrich the expressiveness of their respective LHSs, which usually only allows for local first-order logical conditions, due to missing knowledge of other possible rule applications. Since graph pattern matches already fulfill certain localized structural constraints as defined by their corresponding graph patterns, we do not need to encode those local constraints in the ILP. Thus, building an ILP problem from a limited set of graph pattern matches (i.e., tuples of graph nodes) instead of the whole graph can in most cases reduce the search space of the ILP problem significantly, and, therefore, increase the performance of the ILP optimization step.

The GIPS framework itself consists of four components, which are shown in Figure 1. During build time, the ILP generator component `ILP Gen` and the user API `GIPS API` are generated according to given GIPSL specifications (as indicated with the red star symbol in Figure 1). The `GIPS Core` component interfaces with the incremental GT tool eMoflon to get access to matches and the underlying model data. The gathered set of matches and the model data are used by the generated ILP generator component `ILP Gen` to construct a new ILP problem. Said ILP formulation is then used by the `ILP Adapter` to connect to the ILP solver to supply it with the problem and receive a valid solution (i.e., one that satisfies all constraints) if one exists. Finally, GT rules can be applied to matches by the use of the generated API `GIPS API` based on the solution.

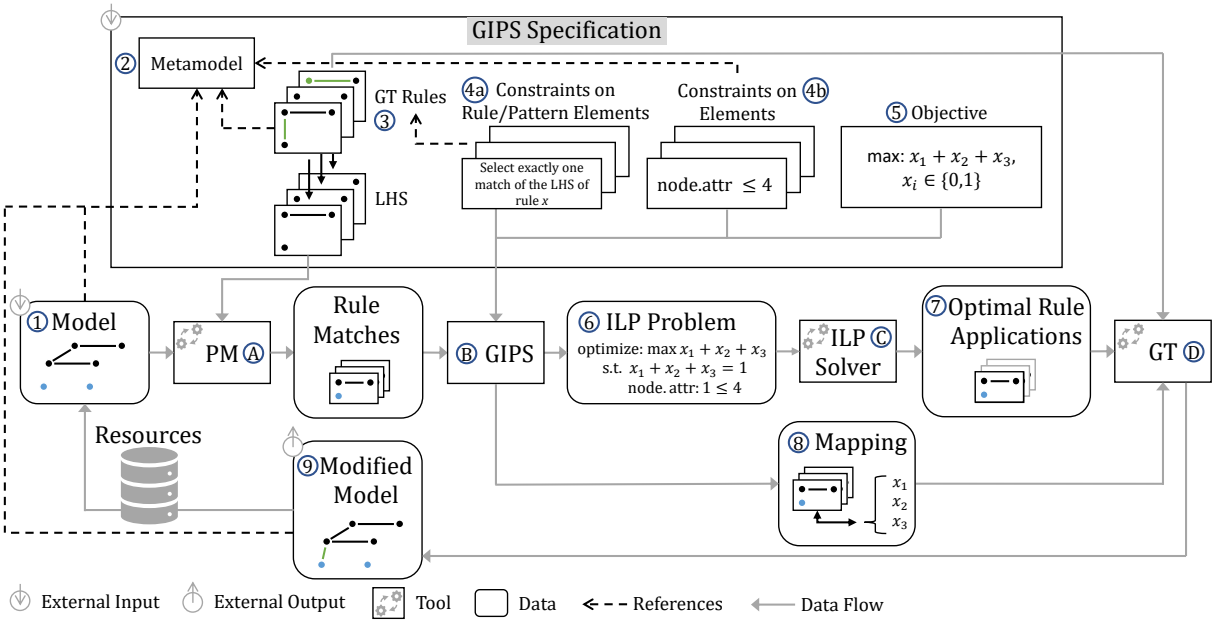


Figure 2: Extended workflow diagram of the GIPS framework [12].

In the following, we will roughly outline how GIPS works, using Figure 2 as an illustration of an exemplary GIPS workflow. Let us assume three input artifacts, namely a metamodel (2), a corresponding graph-based model (1) that is to be altered and the GIPSL specification, which contains a set of GT rules (3), a set of rule and model constraints (4) as well as an objective function (5). The exemplary constraint in (4a) ensures that in the resulting set of GT rule applications, the rule x must be applied exactly once. Furthermore, another exemplary constraint (see (4b)) ensures that an attribute of a model element has to be smaller or equal to 4. The exemplary objective (see (5)) consists of a linear function of ILP decision variables ($x_i \in \{0,1\}$) that is to be maximized. As shown in Figure 2, the GIPS process starts with requesting matches from an external incremental pattern matcher (A). As a result, GIPS has

all valid rule matches and, thus, all locations where a GT rule can be applied. The matches alongside constraints and the objective are used by GIPS (B) to generate an ILP problem (6), where each variable corresponds to a match. In this particular ILP problem example (see (6)), the objective function is taken from (5), the first constraint is the result of (4a), and the second constraint is the result of (4b). Additionally, mappings are created that encode which variable corresponds to which match (8). As previously mentioned, GIPS makes it possible to subject GT rules to ILP constraints, which limit rule applications (4a), e.g., to prevent rules from invalidating matches of other rules LHSs. Besides that, it is also possible to impose ILP constraints onto the model (4b), e.g., to enforce certain (aggregated) attribute values or to limit the total number of edges connected to a node. An external ILP solver (C) will then calculate an optimal solution w.r.t. the given objective function, ensuring all given constraints. Finally, a solution to a given problem implicitly contains valid rule applications (matches) encoded as a set of non-zero binary variables (7). This means that the selected rule applications are in arbitrary order. Using these resulting binary variables in combination with the aforementioned mappings to their corresponding matches and GT rules, an external GT engine (D) performs the graph modifications, which results in a modified model (9).

4 Adapting P2P Overlay Networks for Data Distribution

In the following, we present our example scenario centered around (centralized) P2P overlay network maintenance, to show that a model-driven software engineering approach, as implemented by GIPS, facilitates rapid prototyping of control algorithms. To this end, we demonstrate that we can use GIPS to specify and generate most components of a generic MAPE-K loop [8, 18] (see Figure 4), which is a popular approach for control algorithms of the self-adaptive system's domain. In short, a MAPE-K loop is a sequence of four stages *Monitor*, *Analyze*, *Plan*, and *Execute* over the *Knowledge* base. The latter maintains data on all relevant parts of the system that are needed by the MAPE stages. The *Monitor* collects data from the underlying system and the environment, whereas the *Analyze* stage checks if an adaption of the system is required. If this is the case, it triggers the *Plan* stage to construct an adaption plan. Lastly, the *Execute* stage ensures the plan gets executed to adapt the system.

4.1 Problem Description

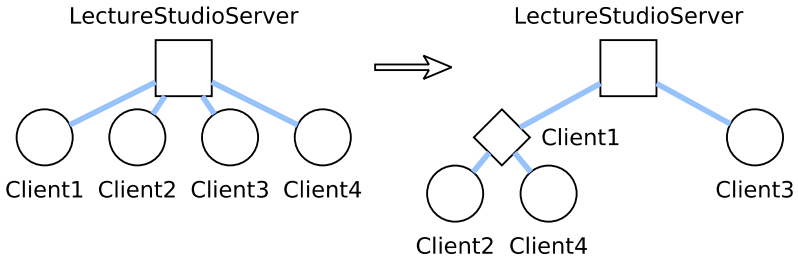


Figure 3: Small example network of a lectureStudio document distribution. The left one is purely centralized and the right one uses a peer-to-peer mechanism. The central server node (`LectureStudioServer`) is shown as a square, relay clients are shown as diamonds, and normal clients are shown as circles.

In recent years, the need for online streaming of lectures and other events has grown considerably. Therefore, the tool and streaming platform lectureStudio has been developed to support lecturers at the Technical University of Darmstadt. One key aspect of this platform is the fact that it is not only able to stream continuous video, e.g., from the webcam of the lecturer or a shared screen but also their slides as a PDF file. The difference to conventional video conference tools is the approach of distributing the used PDF slide set to all participants at the beginning of the lecture. After the lecture has initially started, only small actions such as slide changes or annotation commands are sent to the clients. By using this approach, the total used bandwidth of the streaming process can be reduced drastically. However, as one might notice, the distribution of possibly large PDF files at the beginning of the lecture causes a huge spike in bandwidth demand for the lectureStudio server. In case the document is large or the number of students participating in the stream is high, said operation can saturate the connection of the server for quite some time. To circumvent this issue, the approach presented in this example calculates a tree-based P2P overlay topology to allow client-to-client distribution of the PDF file. By using such a P2P approach, the central lectureStudio server only has to transfer the PDF document to a subset of all connected clients, which will transfer (relay) the file to the remaining clients afterwards.

Figure 3 shows two examples of distribution overlay networks with square and diamond nodes indicating the (re-)distributors of PDF files. The left one uses a classic purely centralized approach in which all clients download the PDF file directly from the server (`LectureStudioServer`). The right network contains a relay client (`Client1`) that forwards the data to two other clients in order to free up some of the server's bandwidth. Hence, our example implementation automatically derives a P2P topology from a given topology and its changes over time, while optimizing a specific objective function to minimize the file distribution time for all participating clients. Therefore, the central algorithm that calculates the P2P overlay topology must determine (1) which clients must become relay clients and (2) how clients connect to other nodes in order to optimize a specific objective function, in this case, to minimize the file distribution time for all clients.

4.2 Prototype Implementation

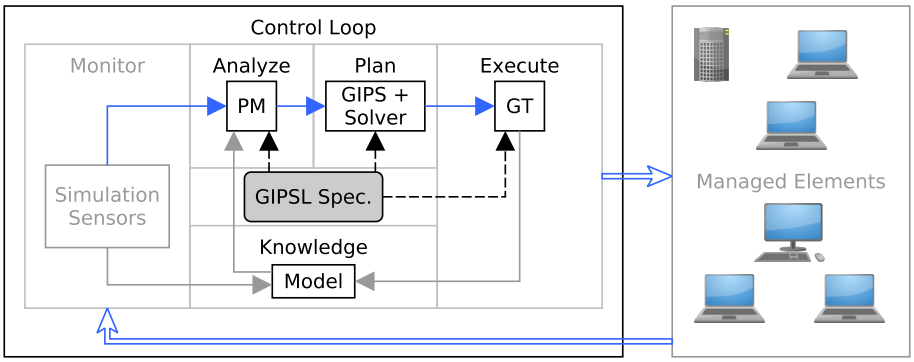


Figure 4: GIPS as MAPE-K loop with the lectureStudio server (control loop) and the clients (managed elements). Blue arrows indicate data flow.

In this section, we present our artifact, which implements an algorithm that solves the challenge described in Section 4.1. Our algorithm is based on a MAPE-K loop as shown in Figure 4, in which the P2P overlay network is incrementally adapted as a reaction to model changes. As indicated in the figure, we used GIPS to generate the building blocks `Analyze`, `Plan`, and `Execute` of the loop from a

high-level GIPSL specification, which makes GIPS an integral part of the submitted artifact. The role of the control loop is played by the lectureStudio server, which coordinates the behavior of all clients that are part of the managed elements. Currently, the whole example and its implementation are only part of a simulation. Hence, all managed elements (clients) only exist as objects in the model and there is no connection to an actual instance of the lectureStudio server application.

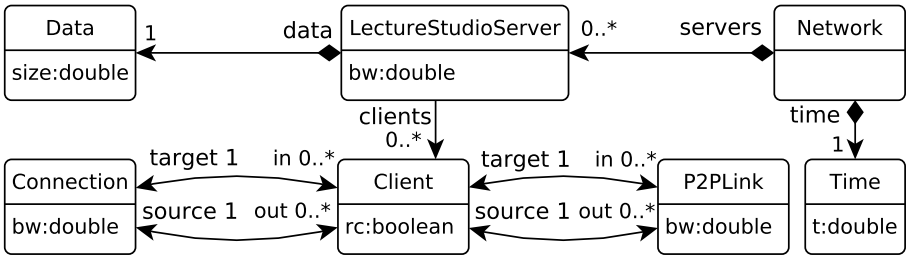


Figure 5: Simplified metamodel of the example's implementation.

Our knowledge component is a graph-based model implemented using the Eclipse Modeling Framework (EMF)[8] (① in Figure 2). This model corresponds to a metamodel (② in Figure 2), which defines the model's node types, attributes, and edges, similar to a UML class diagram. As shown in Figure 5, our network metamodel (which is an example of ② in Figure 2) consists of the following nodes[9]: `Network`, `LectureStudioServer`, `Client`, `Connection`, `P2PLink`, `Data`, and `Time`. A node of type `Network` contains the central node of type `LectureStudioServer`, each containing a set of clients represented by nodes of type `Client`. Nodes of type `Client` can either be normal clients or relay clients; in the latter case, the Boolean flag `rc` of the respective node is set to `true`. Moreover, nodes of type `Connection` connect two nodes of type `Client` via `source` and `target` edges, and model the maximum possible bandwidth with the attribute `bw`. (Virtual) P2P connections are modeled by nodes of type `P2PLink`, which also connect two nodes of type `Client` with an attributed bandwidth `bw`. A `LectureStudioServer` contains one node of type `Data`, which models the PDF file to be transferred when the lecture starts. Finally, a node of type `Time` contains a global variable `t` that is used to calculate the actual number of time steps needed to distribute data to all clients in the simulation. Figure 3 is a simplified example of a network model that corresponds to the metamodel of Figure 5.

To generate the missing blocks of the MAPE-K loop, namely *Analyze*, *Plan*, and *Execute* using GIPS (see Figure 4), we need a GIPSL specification that defines the behavior of said stages through GT rules, patterns, ILP constraints, and an objective function to optimize.

In general, the *Analyze* stage of a MAPE-K loop identifies locations in the model that violate constraints and, in turn, it also detects locations that can be modified to reach a constraint-compliant state again. Moreover, the *Analyze* stage also detects locations that might be used to re-establish an optimal system state (according to the objective function), in case the system has lost this property due to external changes. In our case, the outcome of the *Analyze* stage is largely defined by the LHSs of the GT rules as well as the patterns used in the GIPSL specification (③ in Figure 2). Effectively, the output of this stage is a collection of matches (see the connection of Ⓐ and Ⓑ in Figure 2) each describing a possible location in the input model for the application of a corresponding GT rule, which could possibly repair a constraint violation or improve the objective function value. For example, a specific rule

[8]Eclipse Modeling Framework - `https://www.eclipse.dev/modeling/emf/`

[9]The complete (non-shortened) metamodel can be found in this repository: `https://github.com/Echtzeitsysteme/gips-gcm-2023-example`

can contain instructions to modify the model in such a way that a waiting `Client` will be connected to the `LectureStudioServer` instance, which works towards satisfying the constraint that requires each `Client` in the network being (indirectly) connected to the `LectureStudioServer` node. During runtime, this will instruct the IGPM (Ⓐ in Figure 2) to search the input model for matches of this rule's LHS, which will each contain a `Client` and a `LectureStudioServer` node.

The *Plan* stage uses the matches of the *Analyze* stage along with the constraints (④a and ④b in Figure 2) and the objective function (⑤ in Figure 2) of the GIPSL specification to generate the actual ILP problem (⑥ in Figure 2) that will be solved with the help of an ILP solver (Ⓒ in Figure 2). The overarching goal of the *Plan* stage is to select the best valid rule applications (i.e., rule-match-pairs, ⑦ in Figure 2) from the *Analyze* stage. In our example, we used GIPSL to define 3 GT rules, 8 graph patterns, and 9 constraint specifications, to model the following behaviour: For each newly appearing or vanishing client, our planner tries to select the subset of all possible rule applications that extend or repair the overlay network, such that it minimizes the average bandwidth of all used connections between all nodes while adhering to all specified constraints. Thus, the output of this stage is a set of matches whose corresponding GT rules must be applied in the *Execute* stage. As a side note, GIPS, in general, does not provide a specific order in which the resulting set of rule applications has to be executed. Either users build their own execution logic using the GIPS generated API, or the GIPSL specification is defined in such a way that the execution order does not matter. In our example implementation, we chose the latter option and specified the ILP problem in such a manner that we may execute our set of valid rule applications in an arbitrary order.

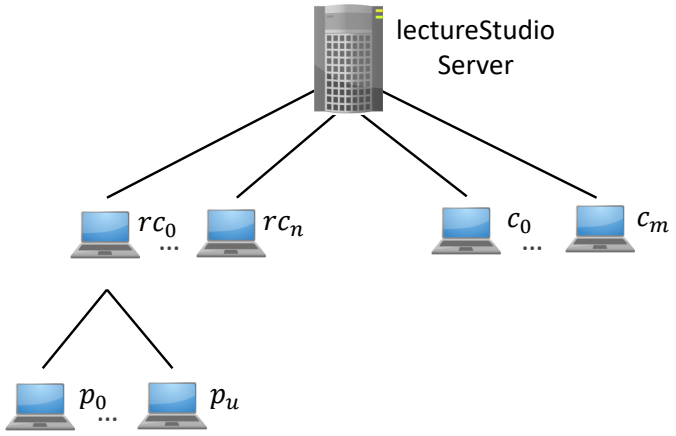


Figure 6: Example network with the following types of nodes: c_0 to c_m are normal clients directly connected to the lectureStudio server, rc_0 to rc_n are relay clients, and p_0 to p_u are relayed clients, i.e., clients that are not directly connected to the lectureStudio server.

As an example of the necessary global constraints used in our lectureStudio scenario, the following paragraph explains three important (simplified) ILP constraints that model inter-rule conditions. This section distinguishes three types of client nodes as shown in Figure 6: c_0 to c_m denote normal clients directly connected to the lectureStudio server, rc_0 to rc_n are relay clients, and p_0 to p_u represent relayed clients, i.e., clients that are not directly connected to the lectureStudio server.

In our approach, every new client with the index i must either be directly connected to the lectureStudio server (c_i), be converted to a relay client (rc_i), or be connected to another node that is a relay client (p_i). The corresponding binary variables (c_i, rc_i, and p_i) are set to 1 if the respective case occurs. The ILP

constraint given in Eq. (2) ensures the explained condition for every new client. Note that *NC* denotes the set of all new clients.

$$\forall i \in \mathbb{N} \text{ with } 0 \leq i < |NC| \;:\; rc_i + c_i + p_i = 1,\; rc_i, c_i, p_i \in \{0,1\} \tag{2}$$

Furthermore, for every (new) relay client with the index i there must be at least one newly added P2P connection to other clients because a regular client should only be converted to a relay client if it actually relays the data to another client. The variable $p2p_{ij}$ denotes a new P2P connection that will be created via a respective GT rule to connect node i and node j. The condition can be expressed with the ILP constraint listed in Eq. (3).

$$\forall i \in \mathbb{N} \text{ with } 0 \leq i < |NC| \;:\; rc_i \leq \sum_{j=0}^{n} p2p_{ij},\; rc_i, p2p_{ij} \in \{0,1\} \text{ and } n \in \mathbb{N}^+ \tag{3}$$

Additionally, it must be ensured that a P2P connection does not use more bandwidth (bw_{ij}) than what is available on the internet link (con_{ij}) between two respective nodes with the indices i and j. This behavior can be achieved with the (simplified) ILP constraint given in Eq. (4), in which P denotes the set of all clients connected to relay clients and RC represents the set of all relay clients.

$$\forall i, j \in \mathbb{N} \text{ with } 0 \leq i < |P| \text{ and } 0 \leq j < |RC| \;:\; x_{ij} \cdot bw_{ij} \leq x_{ij} \cdot con_{ij},\; bw_{ij}, con_{ij} \in \mathbb{N}^+ \tag{4}$$

The complete (and non-simplified) GIPSL definition with all the necessary constraints and the objective of the *Plan* stage can be found on our GitHub repository[10].

The *Monitor* block must currently be implemented by hand and cannot be generated from a specification, despite our generic approach in GIPS. This is due to the fact, that the *Monitor* represents a bridge from the managed system to the knowledge base, which is a highly domain-specific problem and depends on the used technologies. A *Monitor* that is connected to a simulation differs greatly from a *Monitor* that is connected to a server application via the internet.

After a full cycle of the MAPE-K loop, the program is ready to adapt the state of the overlay network again based on the changes that occurred in the meantime. Hence, our example implementation allows for an incremental adaption of the P2P overlay network as a reaction to model changes.

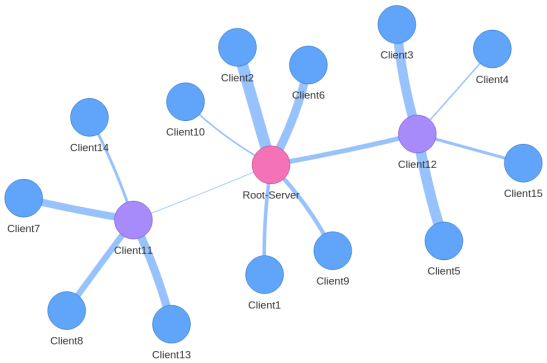
4.3 Prototype Demonstration

We have prepared an example of the calculation of the lectureStudio P2P overlay network. Figure 7 shows screenshots of the resulting network visualizations of the example implementation. The central entity (`Root-Server`) is connected to two relay clients (`Client3` and `Client11`). All normal clients (blue) are either connected to the relay clients (violet) or to the `Root-Server` (pink). The edge thickness denotes the available bandwidth between the nodes. The thicker it is, the more bandwidth is available.

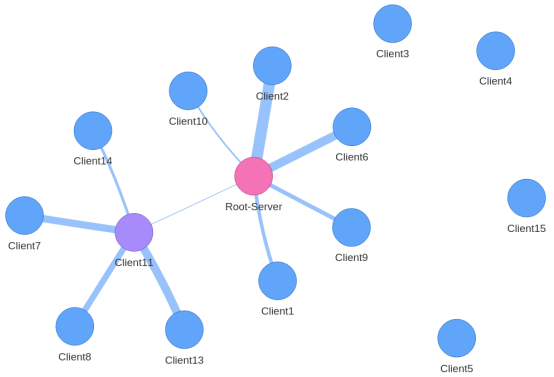
In this scenario, there will be one central `LectureStudioServer` instance and 15 clients in the initial setup (see Figure 7a). As a next step, one of the relay clients will be removed from the model (see Figure 7b), e.g., because the corresponding student left the lecture stream. The incremental implementation is able to formulate the necessary network repair operation as an ILP problem[11] as well and calculates a

[10]GIPSL file for the lectureStudio scenario - `https://github.com/Echtzeitsysteme/gips-gcm-2023-example/blob/main/org.gips.examples.incrementalp2p.gips.incrementaldistribution/src/gipsl/Model.gipsl`

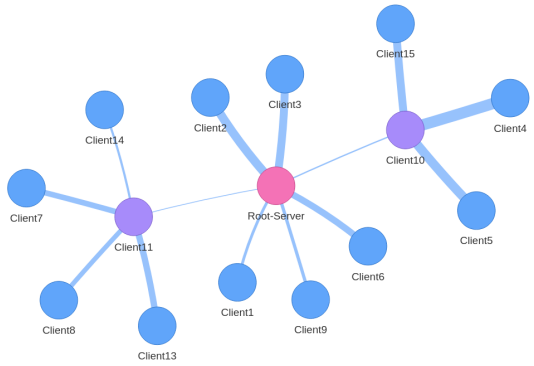
[11]The complete GIPSL definition for all ILP constraints can be found on GitHub - `https://github.com/Echtzeitsysteme/gips-gcm-2023-example/blob/main/org.gips.examples.incrementalp2p.gips.incrementaldistribution/src/gipsl/Model.gipsl`



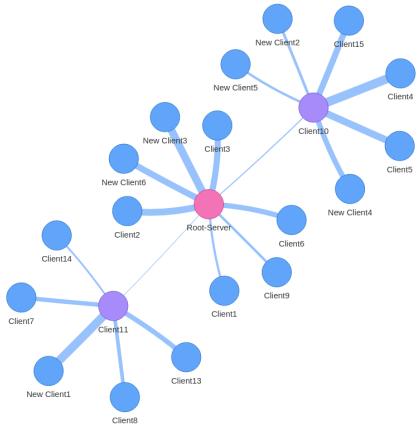
(a) Initial network state.



(b) Second relay client was removed.



(c) Floating clients were reconnected.



(d) Additional clients were connected.

Figure 7: Screenshots of the example's network visualizations.

set of GT rule applications to correct the network, i.e., to re-connect the clients that lost their connection. The result is shown in Figure 7c. To further show the capabilities of the example implementation, the next step adds 6 new clients, that represent, for example, new students joining the stream. The implementation will automatically add these new clients to the data distribution overlay network, which is shown in Figure 7d. The result can be reproduced with the artifact that consists of the GIPS framework and the presented implementation. To ease the installation for interested readers, we created a virtual machine[12] with all required software installed and documentation to be able to modify, execute, and visualize the example. We used the open-source solver GLPK because it does not need any manual licensing. A typical execution of our example finishes after an interval of 2 s to 30 s if run inside the virtual machine. The result of a simulation run will be displayed in the web browser within the virtual machine.

Of course, the question of how well the presented approach scales w.r.t. the number of clients within the streaming network arises. To answer this question, we ran experiments with different numbers of clients that incrementally join the stream. In this scenario, our generated topology control algorithm connects all newly arriving clients to the P2P network, while minimizing the overall document distri-

[12]GIPS GCM 2023 Artifact VM - `https://github.com/Echtzeitsysteme/gips-gcm-2023-artifact-vm`

bution time. In the evaluation setup, the `LectureStudioServer` is connected to the network via a 150 Mbit/s connection and no initial `Client` has joined, yet. The upload and download bandwidths of all `Clients` are sampled from Speedtest.net measurements[13]. Every result shown in this section is the calculated mean of ten measurements. All experiments were run on a workstation equipped with an AMD Ryzen Threadripper 2990WX with 32 CPU cores and 128 GB of memory. The operating system used is Ubuntu 20.04.6 LTS, the Java environment used is OpenJDK Temurin (build 17.0.2+8), and the ILP solver is GLPK. Figure 8 shows runtimes (y-axis) of the algorithm for each individual number of clients (x-axis), additionally, separated into GT, ILP, and miscellaneous parts. The results show that the generated algorithm calculates a P2P topology for up to 75 clients in under one minute. Unfortunately, Figure 8 shows a large growth of the runtime when scaling up the number of `Clients`. Interestingly, the majority of the runtime needed to calculate each result is caused by the pattern-matching process of the GT engine. Thus, the time needed by GLPK to solve each resulting ILP problem is barely visible. The exponential increase in runtime is most likely a result of the used graph patterns. Some of the patterns contain multiple disjoint pattern nodes, which results in match sets that grow exponentially with an increasing number of clients.

When using GIPS in practice, sometimes the GT part and sometimes the ILP part dominate the runtime behavior. The sum of both is often lower than the runtime of a purely ILP-based approach. Corresponding comparisons of naive generation of the ILP problem with GT rules with very low GT runtime and almost the entire runtime of the ILP solver, as well as a heavier preprocessing by GT rules, are planned for a future publication.

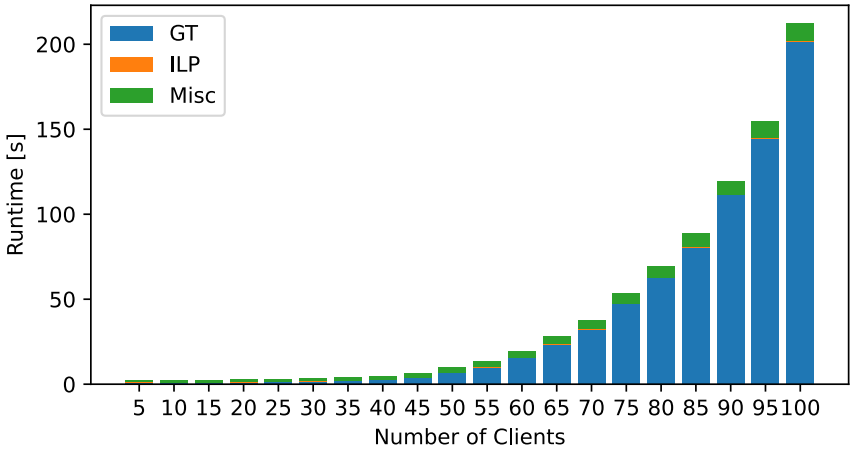


Figure 8: Aggregated algorithm runtime (y-axis) for an individual number of clients (x-axis) incrementally joining a lecture. The individual runtimes are split into GT, ILP, and miscellaneous parts.

4.4 Challenges in leaving the Prototype Phase

In the previous sections, we have shown how to use GIPS as a tool for prototyping a control algorithm using an example of the topology control domain. More specifically, we have used GIPS to automatically derive MAPE-K building blocks from a GIPSL specification. From a rapid prototyping point of view, one can see that GIPS is an excellent tool to facilitate the development of (topology) control algorithms,

[13] Speedtest.net median bandwidth in Germany - `https://www.speedtest.net/global-index/germany`

since GIPS can be used to develop and evaluate prototypes by experimenting with parameters, different objective functions, and different constraints.

A possible design challenge is the fact that the result of a GIPS calculation is a set of GT rule matches without any additional information about a suitable execution order that would guarantee that all selected matches are processed. To guarantee that a GT rule application does not invalidate a match of another GT rule, developers must introduce additional GIPSL constraints, if this is a necessary requirement for the specific problem domain.

While GIPS is a promising approach for developing prototypes, it is clear that a control algorithm based on ILP and GT will not be suitable for most real-world application scenarios that have to work under real-time constraints (e.g., streaming, content distribution, etc.). However, the developed prototypes can be used to calculate solutions for offline optimization problems, e.g., static task scheduling. Furthermore, there are other application domains, for example, the placement of virtual networks in data centers, for which GIPS with its ILP and GT-based algorithms can compete with state-of-the-art approaches [12]. In such a problem domain, the generated algorithms scale reasonably well and, moreover, the developers are able to try and evaluate different heuristic approaches, too. Once a GIPS-based prototype is developed far enough to satisfy the desired specifications, one can use the results of the prototype to create an optimized implementation using, e.g., the C++ programming language for further acceleration. In the Machine Learning (ML) community the approach of programmatic labeling (programmatic weak supervision) is well known, where data sets are automatically annotated for training ML using exact algorithms, heuristics, or other ML approaches [22]. We propose to implement an exact algorithm using our GIPS framework to annotate data sets for an ML approach, e.g., neural networks, automatically. This means that we take automatically generated problem instances and solve them using the GIPS algorithm to convert the obtained results to a training data set. In fact, this is an approach we are currently pursuing and will present in a future paper.

5 Related Work

We have identified a general need for model-driven rapid prototyping because of the ever-growing complexity of modern software systems. To be able to relate GIPS to the current state-of-the-art, we will list a few related works of the self-adaptive systems community as well as selected works of the software engineering community, which touch on the same subjects as our work such as, e.g., graph transformation, optimization, MAPE-K loop development, and resource allocation.

Becker et al. [4] defined the requirements to model correct self-adaptive systems on a high level of abstraction and presented a GT-based formal solution to most of the identified requirements. The authors explain how UML class and object diagrams together with the behavioral modeling of GT can be used to model correct self-adaptive systems. This concept is evaluated using a simplified application example with a network of pipes and filters. Similar to our approach in GIPS, Becker et al. used GT to express local structural constraints and model transitions, while GIPS, on the other hand, is also able to express and enforce global constraints on the model by using ILP.

Abeywickrama et al. [2] presented a novel approach to engineer interacting, centralized, and decentralized feedback loops using the state of the affairs model [1] to express self-awareness and self-adaption. Therefore, their notion of feedback loops extends the well-known MAPE-K adaption model [8, 18]. In their work, they implemented an Eclipse plug-in for the modeling, simulation, and validation of self-adaptive systems based on their feedback-loop-based approach. Compared to GIPS, their Eclipse plug-in is currently unable to derive runtime code (e.g., Java) from the models.

Fleck et al. [14] presented a model-driven approach to tackle the rule orchestration problem. They try to find an ordered set of rule-match pairs to modify a model to optimize a given fitness function. In contrast to GIPS, the result is a multi-objective optimization problem intended to be solved by a search-based optimization algorithm, for example, a genetic algorithm. A downside of this approach is the fact that genetic algorithms cannot guarantee optimal solutions and, furthermore, can have problems finding promising sequences of rule applications in the large search space. If runtime is not an issue, GIPS can guarantee optimal solutions by utilizing the ILP solver. In contrast to GIPS, the approach by Fleck et al. does not need annotated rules or the specification of additional constraints to find the rule application sequence.

With the framework MDEOptimiser Burdusel et al. [9] present another search-based solution to find sequences of rule applications that, when applied, lead to an optimal model state. In contrast to [14], which performs rule-based optimization, Burdusel et al. follow the model-based optimization approach. In essence, they store each evolution of the initial model as a self-contained new model during the optimization process, instead of only storing the changes (i.e., rule applications) between each optimization step. Similar to [14], Burdusel et al. cannot guarantee optimal solutions, if they exist, due to the usage of evolutionary algorithms as means of optimization.

John et al. [17] make use of MDEOptimiser to demonstrate and practically evaluate their new formal framework for model-based optimizations, in which they define completeness and soundness as new criteria for mutation operators.

Chen et al. [10] implement an incremental generative self-adaptation scheme based on model transformations at runtime. Similar to our approach, the authors implement a MAPE-K loop, where a planning component produces a set of GT rule applications that, when applied, improve the utility of a given model. In contrast to our approach, Chen et al. make use of Satisfiability Modulo Theories (SMT) to define their repair problem, use SMT solvers to solve said problem, and implement their model transformations using medini QVT. The former design choice makes it very hard to define a problem that, when solved, results in a solution that is not only valid (satisfies all constraints) but also optimal according to some objective function. The latter design choice renders the tool unusable today since medini QVT was deprecated a while ago.

Song et al. [20] do not present a self-adaption approach per se but, instead, present a framework similar to GIPS, with which one can implement algorithms that apply changes to a given model such that it satisfies a set of constraints. Similar to Chen et al. [10], Song et al. make use of SMT to encode model constraints. Furthermore, their approach comes with a very lean DSL, mostly for configuration purposes, where model constraints can be defined using OCL expressions. In contrast to GIPS, their approach does not give the means to define any model transformations with which a model could be changed, in order to satisfy a set of constraints.

What Ghahremani et al. [16] describe as utility-driven self-healing for dynamic architectures is, essentially, an approach for incremental model adaptation at runtime, which is in a way similar to Chen et al. [10]. In contrast to Chen et al. and our approach, Ghahremani et al. do not specify an SMT nor an ILP problem; instead, they implement a search-based approach that aims to improve the result value of a given utility function. Said utility function is composed of several (possibly user-defined) sub-utility functions that each evaluate matches belonging to the LHS of a corresponding "repair" rule. The goal is to find a set of rule applications that maximize the utility function and improve or "heal" the given model.

6 Conclusion and Future Work

This paper presented an approach for the rapid prototype development of control algorithms, aided by GT, ILP, and code generation from a high-level specification. Therefore, we briefly introduced the GIPS framework and explained an example algorithm for the maintenance of P2P overlay networks, which was, for the most part, generated by GIPS from a high-level specification, in an effort to demonstrate GIPS's capabilities as a rapid prototyping software development tool. To provide these capabilities, our framework was designed with MDSE principles in mind and, thus, generates software artifacts from a given GIPSL specification. With our example of the P2P overlay network maintenance tool for lectureStudio, we demonstrated the automatic derivation of building blocks for control algorithms to support the development of MAPE-K loops. By means of ILP-based optimization our algorithm sets up and maintains an overlay network for the streaming platform lectureStudio with the goal to minimize the required bandwidth for file transfers while upholding an acceptable document distribution time for all participants. From our point of view, this example demonstrates the usefulness of the GIPS framework for rapid prototyping to facilitate research within the domain of (topology) control algorithms.

In the future, we want to gather more information on how GIPS can be used in research of various (possibly more complex) domains, by implementing and evaluating a variety of other scenarios, e.g., the static scheduling of tasks on CPU processors. Regarding the control algorithm domain, it is still an open question how or if the *Monitor* block of the MAPE-K loop can be generated automatically. Moreover, it would be interesting to include a mechanism to support the generation of valid GT rule application sequences (instead of sets) to guarantee the validity of the GIPS output. Finally, as an ongoing effort, we plan to extend the expressiveness of GIPSL further, by developing new language features and shortcuts to ease the specification process for tool designers of various domains.

The presented example containing the GIPS framework, an open-source ILP solver, the algorithm's source code, and its documentation, is publicly available on GitHub[14] as a virtual machine.

Acknowledgements This work has been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - Project-ID 210487104 - SFB 1053.

References

[1] Dhaminda B. Abeywickrama, Nicola Bicocchi & Franco Zambonelli (2012): *SOTA: Towards a General Model for Self-Adaptive Systems*. In: *2012 IEEE 21st Int. Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pp. 48–53, doi:10.1109/WETICE.2012.48.

[2] Dhaminda B. Abeywickrama, Nicklas Hoch & Franco Zambonelli (2013): *SimSOTA: Engineering and Simulating Feedback Loops for Self-Adaptive Systems*. In: *Proc. of the Int. C* Conf. on Computer Science and Software Engineering*, C3S2E '13, Association for Computing Machinery, p. 67–76, doi:10.1145/2494444.2494446.

[3] Mokhtar S. Bazaraa, John J. Jarvis & Hanif D. Sherali (2011): *Linear programming and network flows*. John Wiley & Sons, doi:10.1002/9780471703778.

[4] Basil Becker & Holger Giese (2008): *Modeling of Correct Self-Adaptive Systems: A Graph Transformation System Based Approach*. In: *Proc. of the 5th Int. Conf. on Soft Computing as Transdisciplinary Science and Technology*, CSTST '08, Association for Computing Machinery, p. 508–516, doi:10.1145/1456223.1456326.

[14] GIPS GCM 2023 Artifact VM - `https://github.com/Echtzeitsysteme/gips-gcm-2023-artifact-vm`

[5] Nelly Bencomo, Sebastian Götz & Hui Song (2019): *Models@run.time: a guided tour of the state of the art and research challenges*. Software & Systems Modeling 18(5), pp. 3049–3082, doi:10.1007/s10270-018-00712-x.

[6] Gordon Blair, Nelly Bencomo & Robert B. France (2009): *Models@ run.time*. Computer 42(10), pp. 22–27, doi:10.1109/MC.2009.326.

[7] Stephen P. Bradley, Arnoldo C. Hax & Thomas L. Magnanti (1977): *Applied Mathematical Programming*. Addison-Wesley.

[8] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè & Mary Shaw (2009): *Engineering Self-Adaptive Systems through Feedback Loops*, pp. 48–70. Springer, doi:10.1007/978-3-642-02161-9_3.

[9] Alexandru Burdusel, Steffen Zschaler & Daniel Strüber (2018): *MDEoptimiser: A Search Based Model Engineering Tool*. In: Proc. of the 21st ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS '18, Association for Computing Machinery, p. 12–16, doi:10.1145/3270112.3270130.

[10] Bihuan Chen, Xin Peng, Yijun Yu, Bashar Nuseibeh & Wenyun Zhao (2014): *Self-Adaptation through Incremental Generative Model Transformations at Runtime*. In: Proc. of the 36th Int. Conf. on Software Engineering, ICSE 2014, Association for Computing Machinery, p. 676–687, doi:10.1145/2568225.2568310.

[11] Betty H. C. Cheng, Kerstin I. Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi A. Müller, Patrizio Pelliccione, Anna Perini, Nauman A. Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann & Norha M. Villegas (2014): *Using Models at Runtime to Address Assurance for Self-Adaptive Systems*, pp. 101–136. Springer, doi:10.1007/978-3-319-08915-7_4.

[12] Sebastian Ehmes, Maximilian Kratz & Andy Schürr (2022): *Graph-Based Specification and Automated Construction of ILP Problems*. In: Proc. of the Thirteenth Int. Workshop on Graph Computation Models, Nantes, France, 6th July 2022, Electronic Proceedings in Theoretical Computer Science 374, Open Publishing Association, pp. 3–22, doi:10.4204/EPTCS.374.3.

[13] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange & Gabriele Taentzer (2006): *Fundamentals of Algebraic Graph Transformation*. Springer, doi:10.1007/3-540-31188-2.

[14] Martin Fleck, Javier Troya & Manuel Wimmer (2016): *Search-based model transformations*. Journal of Software: Evolution and Process, pp. 1081–1117, doi:10.1002/smr.1804.

[15] Charles L. Forgy (1982): *Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem*. Artificial Intelligence, p. 17–37, doi:10.1016/0004-3702(82)90020-0.

[16] Sona Ghahremani, Holger Giese & Thomas Vogel (2020): *Improving Scalability and Reward of Utility-Driven Self-Healing for Large Dynamic Architectures*. ACM Trans. Auton. Adapt. Syst. 14(3), doi:10.1145/3380965.

[17] Stefan John, Jens Kosiol, Leen Lambers & Gabriele Taentzer (2023): *A graph-based framework for model-driven optimization facilitating impact analysis of mutation operator properties*. Software and Systems Modeling, doi:10.1007/s10270-022-01078-x.

[18] Jeffrey O. Kephart & David M. Chess (2003): *The vision of autonomic computing*. Computer 36(1), pp. 41–50, doi:10.1109/MC.2003.1160055.

[19] David G. Luenberger & Yinyu Ye (1984): *Linear and Nonlinear Programming*. Springer, doi:10.1007/978-3-319-18842-3.

[20] Hui Song, Xiaodong Zhang, Nicolas Ferry, Franck Chauvel, Arnor Solberg & Gang Huang (2014): *Modelling Adaptation Policies as Domain-Specific Constraints*. In: Model-Driven Engineering Languages and Systems, Springer, pp. 269–285, doi:10.1007/978-3-319-11653-2_17.

[21] Stefan Tomaszek, Roland Speith & Andy Schürr (2021): *Virtual network embedding: ensuring correctness and optimality by construction using model transformation and integer linear programming techniques*. Software and Systems Modeling, pp. 1299–1332, doi:10.1007/s10270-020-00852-z.

[22] Jieyu Zhang, Cheng-Yu Hsieh, Yue Yu, Chao Zhang & Alexander Ratner (2022): *A Survey on Programmatic Weak Supervision*. *CoRR* abs/2202.05433, doi:10.48550/arXiv.2202.05433.